2A Symbolics Common Lisp-Language Concepts

symbolics

2A Symbolics Common Lisp–Language Concepts

symbolics™

Cambridge, Massachusetts

Symbolics Common Lisp: Language Concepts

# 999018

August 1986

This document corresponds to Genera 7.0 and later releases.

Symbolics, Inc.
4 New England Tech Center
555 Virginia Road
Concord, MA 01742

Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.
Text masters produced on Symbolics 3600-family computers and printed on Symbolics LGP2 Laser Graphics Printers.
Cover Design: Schafer|LaCasse
Printer: CSA Press
Printed in the United States of America.

Printing year and number: 88 87 86 9 8 7 6 5 4 3 2 1

Table of Contents

List of Figures

Preface

Together, the two volumes of this book are your guide to Symbolics Common Lisp (SCL), the Symbolics implementation of the Lisp language.

Lisp is a powerful and complex tool that can be used at many levels, by people with widely varying programming experience. The Symbolics implementation of Lisp in the Genera software environment is therefore intended to serve a user spectrum that ranges from the novice programmer to the experienced Lisp developer. These two facts motivate the organization of this book into three major parts, each reflecting a different stage of familiarity with Lisp.

Volume A contains two related, but distinct presentations of the concepts you need to understand in order to put Symbolics Common Lisp to the fullest possible use. The first of these, Overview of Symbolics Common Lisp, is intended primarily as a learning aid – to give the new user an introduction to key SCL concepts. The Overview does not present topics in any detail. Rather, it is aimed at giving the new user a general sense of each topic, including definitions of basic terms and simple examples of important concepts. It is designed to be read sequentially, in a single sitting if desired.

The second component of Volume A is a much deeper conceptual presentation of Symbolics Common Lisp, called Language Concepts. Language Concepts provides in-depth coverage of topics presented in the Overview, as well as a few advanced topics not covered in the Overview. The order of topics in Language Concepts parallels that of the Overview. Unlike the overview, however, it is intended to be used as either conceptual or reference documentation.

Volume B contains the third, and most detailed, part of this document, the Language Dictionary. This is a true dictionary of reference entries for all Symbolics Common Lisp symbols. Each entry provides a complete description of a single Lisp object. For example, the entry for a given SCL function would include its syntax, what it returns, examples of its use and cross-references to related functions or topics. The entries are alphabetized and thumb tabs are provided for rapid access to information about an individual symbol when you need it. Because the dictionary entries in Volume B are in alphabetical order, this volume of Symbolics Common Lisp is not indexed; Volume A is fully indexed.

Understanding Notation Conventions

You should understand several notation conventions before reading the documentation.

Lisp Objects

Functions

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3 (arg4* **(foo3)***)* *function*

Adds together *arg1* and *arg2*, and then multiplies the result by *arg3*. If *arg3* is not provided, the multiplication is not done. **function-name** returns a list whose first element is this result and whose second element is *arg4*. Examples:

```
(function-name 3 4) => (7 4)
(function-name 1 2 2 'bar) => (6 bar)
```

Winston/ Horn 122/123

The word "&optional" in the list of arguments tells you that all of the arguments past this point are optional. The default value of an argument can be specified explicitly, as with *arg4*, whose default value is the result of evaluating the form **(foo 3)**. If no default value is specified, it is the symbol **nil**. This syntax is used in lambda-lists in the language. (For more information on lambda-lists: See the section "Evaluating a Function Form", page 504.) Argument lists can also contain "&rest", which is part of the same syntax.

Note that the documentation uses several *fonts*, or typefaces. In a function description, for example, the name of the function is in boldface in the first line, and the arguments are in italics. Within the text, printed representations of Lisp objects are in the same boldface font, such as **(+ foo 56)**, and argument references are italicized, such as *arg1* and *arg2*.

Other fonts are used as follows:

"Typein" or "example" font (`function-name`)

Indicates something you are expected to type. This font is also used for Lisp examples that are set off from the text and in some cases for information, such as a prompt, that appears on the screen.

"Key" font (RETURN, c-L)

For keystrokes mentioned in running text.

Macros and Special Forms

The descriptions of special forms and macros look like the descriptions of these imaginary ones:

do-three-times *form* *Special Form*

Evaluates *form* three times and returns the result of the third evaluation.

with-foo-bound-to-nil *form...* *Macro*

Evaluates the *forms* with the symbol **foo** bound to **nil**. It expands as follows:

```
(with-foo-bound-to-nil
form1
form2 ...) ==>
(let ((foo nil))
form1
form2 ...)
```

Since special forms and macros are the mechanism by which the syntax of Lisp is extended, their descriptions must describe both their syntax and their semantics; unlike functions, which follow a simple consistent set of rules, each special form is idiosyncratic. The syntax is displayed on the first line of the description using the following conventions.

- Italicized words are names of parts of the form that are referred to in the descriptive text. They are not arguments, even though they resemble the italicized words in the first line of a function description.

- Parentheses ("( )") stand for themselves.

- Brackets ("**[ ]**") indicate that what they enclose is optional.

- Ellipses ("**...**") indicate that the subform (italicized word or parenthesized list) that precedes them can be repeated any number of times (possibly no times at all).

- Braces followed by ellipses ("{ }**...**") indicate that what they enclose can be repeated any number of times. Thus, the first line of the description of a special form is a "template" for what an instance of that special form would look like, with the surrounding parentheses removed.

The syntax of some special forms is too complicated to fit comfortably into this style; the first line of the description of such a special form contains only the name, and the syntax is given by example in the body of the description.

The semantics of a special form includes not only its contract, but also which subforms are evaluated and what the returned value is. Usually this is clarified with one or more examples.

A convention used by many special forms is that all of their subforms after the first few are described as "*body*...". This means that the remaining subforms constitute the "body" of this special form; they are Lisp forms that are evaluated one after another in some environment established by the special form.

This imaginary special form exhibits all of the syntactic features:

twiddle-frob *[(frob option...)] {parameter value}...* *Special Form*

Twiddles the parameters of *frob*, which defaults to **default-frob** if not specified. Each *parameter* is the name of one of the adjustable parameters of a frob; each *value* is what value to set that parameter to. Any number of *parameter/value* pairs can be specified. If any *options* are specified, they are keywords that select which safety checks to override while twiddling the parameters. If neither *frob* nor any *options* are specified, the list of them can be omitted and the form can begin directly with the first *parameter* name.

frob and the *values* are evaluated; the *parameters* and *options* are syntactic keywords and are not evaluated. The returned value is the frob whose parameters were adjusted. An error is signalled if any safety check is violated.

Flavors, Flavor Operations, and Init Options

Flavors themselves are documented by the name of the flavor.

Flavor operations are described in three ways: as methods, as generic functions, and as messages. When it is important to show the exact flavor for which the method is defined, methods are described by their function specs. Init options are documented by the function spec of the method.

When a method is implemented for a set of flavors (such as all streams), it is documented by the name of message or generic function it implements.

The following examples are taken from the documentation.

sys:network-error *Flavor*

This set includes errors signalled by networks. These are generic network errors that are used uniformly for any supported networks. This flavor is built on **error**.

:clear-window of **tv:sheet** *Method*

Erase the whole window and move the cursor position to the upper left corner of the window.

:tyo *char* *Message*

The stream will output the character *char*. For example, if **s** is bound to a stream, then the following form will output a "B" to the stream:

```
(send s :tyo #\B)
```

For binary output streams, the argument is a nonnegative number rather than specifically a character.

dbg:special-command-p *condition special-command* *Generic Function*

Returns **t** if *command-type* is a valid Debugger special command for this condition object; otherwise, returns **nil**.

The compatible message for **dbg:special-command-p** is:

:special-command-p

For a table of related items: See the section "Basic Condition Methods and Init Options", page 599.

:bottom *bottom-edge* (for **tv:sheet**) *Init Option*

Specifies the y-coordinate of the bottom edge of the window.

Variables

Descriptions of variables ("special" or "global" variables) look like this:

typical-variable *Variable*

The variable **typical-variable** has a typical value....

Macro Characters

Macro characters are explained in detail in the documentation. See the section "How the Reader Recognizes Macro Characters" in *Reference Guide to Streams, Files, and I/O.*

The single quote character (') and semicolon (;) have special meanings when typed to Lisp; they are examples of what are called *macro characters.* It is important to understand their effect.

When the Lisp reader encounters a single quote, it reads in the next Lisp object and encloses it in a **quote** special form. That is, **'foo-symbol** turns into **(quote foo-symbol)**, and **'(cons 'a 'b)** turns into **(quote (cons (quote a) (quote b)))**. The reason for this is that "**quote**" would otherwise have to be typed in very frequently and would look ugly.

In Lisp, *quoting* a character means inhibiting what would otherwise be special processing of it. Thus, in Common Lisp, the backslash character, "\", is used for quoting unusual characters so that they are not interpreted in their usual way by the Lisp reader, but rather are treated the way normal alphabetic characters are treated. So, for example, in order to give a "\" to the reader, you must type "\\", the first "\" quoting the second one. When a character is preceded by a "\" it is said to be *slashified.* Slashifying also turns off the effects of macro characters such as single quote and semicolon. Note that in Zetalisp syntax, the slash, "/", is the quoting character and must be doubled.

The following characters also have special meanings, and cannot be used in symbols without slashification. These characters are explained in detail elsewhere: See the section "How the Reader Recognizes Macro Characters" in *Reference Guide to Streams, Files, and I/O.*

"	Double-quote delimits character strings.
#	Sharp-sign introduces miscellaneous reader macros.
‘	Backquote is used to construct list structure.
,	Comma is used in conjunction with backquote.
:	Colon is the package prefix.
|	Characters between pairs of vertical bars are quoted.
⊗	Circle-X lets you type in characters using their octal codes. (Zetalisp only.)

The semicolon is used as a commenting character. When the Lisp reader sees one, the remainder of the line is ignored.

Character Case

All Lisp code in the documentation is written in lowercase. In fact, the Lisp reader turns all symbols into uppercase, and consequently everything prints out in uppercase. You can write programs in whichever case you prefer.

Packages and Keyword Names

For an explanation of packages: See the section "Packages", page 635.

Various symbols have the colon (:) character in their names. By convention, all *keyword symbols* in the system have names starting with a colon. The colon character is not actually part of the print name, but is a truncated package prefix indicating that the symbol belongs to the **keyword** package. (For more information on colons: See the section "Introduction to Keywords", page 128. For now, just pretend that the colons are part of the names of the symbols.)

The document set describes a number of internal functions and variables, which can be identified by the "**si:**" prefix in their names. The "**si**" stands for "**system-internals**". These functions and variables are documented because they are things you sometimes need to know about. However, they are considered internal to the system and their behavior is not as guaranteed as that of everything else.

Maclisp

Because Symbolics Common Lisp is descended from Maclisp, some Symbolics Common Lisp functions exist solely for Maclisp compatibility; they should *not* be used in new programs. Such functions are clearly marked in the text.

Examples

The symbol "=>" indicates Lisp evaluation in examples. Thus, "**foo** => **nil**" means the same thing as "the result of evaluating **foo** is **nil**".

The symbol "==>" indicates macro expansion in examples. Thus, "**(foo bar)** ==> **(aref bar 0)**" means the same thing as "the result of expanding the macro **(foo bar)** is **(aref bar 0)**".

The Character Set

The Genera character set is not the same as the ASCII character set used by most operating systems. For more information: See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O.*

Unlike ASCII, there are no "control characters" in the character set; Control and Meta are merely things that can be typed on the keyboard.

PART I.

Overview of Symbolics Common Lisp

The following sections introduce the topics covered in the rest of this manual. The aim is to provide you with a very general sense of the basic concepts and terms in Symbolics Common Lisp, in a form that you can read at a single sitting. If you are a new user you might find this material of particular interest.

The Lisp dialect documented here is Symbolics Common Lisp. Symbolics Common Lisp is based on Common Lisp, and includes Common Lisp, as well as all the advanced features of Zetalisp. Details about the relationship between these three dialects are provided elsewhere: See the section "Introduction to Symbolics Common Lisp", page 671.

The individual chapters in See the document *Symbolics Common Lisp: Language Concepts.* offer more extended coverage of topics appearing in this Overview; the alphabetized dictionary provides full information about individual functions and other Lisp objects.

General information about two topics, Cells and Locatives and Special Forms, appears exclusively in this Overview, the former because the topic does not require further coverage, the latter because special forms are scattered throughout "Symbolics Common Lisp" and are covered in the context of various other topics. See the section "Cells and Locatives", page 29. See the section "Special Forms and Built-in Macros", page 35.

The term *form* is ubiquitous in any discussion of the Lisp language and so is worth mentioning here. A *form* is a data object that is meant to be evaluated.

1. Overview of Data Types

1.1 Overview of Data Types and Type Specifiers

Lisp is a *typed* language; Lisp programs manipulate data structures of a given *type*, using them to build more complex structures. The term *Lisp object* refers to the collectivity of basic data types that programs can create. Some examples of Lisp objects are symbols, characters, and structure and flavor instances.

Symbolics Common Lisp provides a wide variety of data object types, as well as facilities for extending the type hierarchy. It is important to note that in Lisp it is data objects that are typed, not variables. Any variable can have any Lisp object as its value.

In Symbolics Common Lisp, a data type is a (possibly infinite) set of Lisp objects. The defined data types are arranged into a hierarchy (actually a partial order) defined by the subset relationship.

A type called **common** encompasses all the data types required by the Common Lisp language. Symbolics Common Lisp provides several additional data types, such as *flavors*, that represent an extension to the Common Lisp type system. The set of all objects in Symbolics Common Lisp is specified by the symbol **t**. The empty data type, which contains no objects, is denoted by **nil**.

The type hierarchy can be conceptualized as a tree whose root is **t**. The following terminology is useful for expressing the basic relationships among the branches and sub-branches of this tree.

A given type is a *supertype* of those data types it encompasses. For example, the type **number** is a *supertype* of all other numeric types such as *rational*, *integer*, *complex*, and so on. These numeric types are called *subtypes* of **number**. They can in turn have supertype-subtype relationships with each other; for example, the type **rational** is a *supertype* of the type **integer**, which is a *supertype* of the type **signed-byte**, and so forth.

The type **t** is a supertype of every type whatsoever: every object belongs to type **t**. The type **nil** is a subtype of every type whatsoever: no object belongs to type **nil**.

Two or more data types can be *disjoint*, that is no object can simultaneously belong to more than one of these types. For example, the types **float** and **rational** are *disjoint subtypes* of the type **number**.

If several types are subtypes of a common supertype, they form an *exhaustive union*. For example, the type **common** denotes an *exhaustive union* of, among others, the types *cons*, *symbol*, *readtable*, *pathname*, and all types created by the user with **defstruct** or **defflavor**. If the types belonging to a common supertype are disjoint, they form an *exhaustive partition.* For example, the types **bignum** and **fixnum** form an *exhaustive partition* of the type **integer**, since an integer can be either a *fixnum* or a *bignum.* For a complete list of Symbolics Common Lisp data types: See the section "Hierarchy of Data Types", page 69.

Data objects such as numbers or arrays are identified by symbolic names or lists, called *type specifiers*, that are associated with them. Type specifiers serve as arguments to predicates that perform type-checking; they are also used by various functions whose operation requires arguments or results of a specific data type.

Examples of some major type specifier symbols are *number*, *character*, *list*, *array*, *table*, *flavor*, and *generic function.* These and many others are discussed in individual chapters in the documentation.

Type specifier lists let you refine type distinctions and define your own types. For example, the type specifier list `(integer` *low high*`)` lets you define an integer type whose range is restricted to the limits indicated by the arguments *low* and *high.*

Since many Lisp objects belong to more than one group of data types, it does not always make sense to ask what *the* type of an object is; instead, one usually asks only whether an object *belongs* to a given type. The predicate **typep** tests a Lisp object against one of the standard type specifiers to determine if it belongs to that type. See the section "Type-checking Differences Between Symbolics Common Lisp and Zetalisp", page 80.

Other basic operations with data types are converting an object of one type to an equivalent object of another type (**coerce**), testing relationships between objects in the type hierarchy (**subtypep**), determining a type to which an object belongs (**type-of**), getting the type specifier list for standard data types (**sys:type-arglist**), and identifying equivalent data type descriptions (**equal-typep**).

1.2 Overview of Numbers

Symbolics Common Lisp includes several types of numbers, with different characteristics. These are:

- Rational Numbers. Used for exact mathematical calculations. These include:
  - *Integers.* These are rational numbers without a fractional part, such as 0, 1, 2.
  - *Ratios.* A ratio is a pair of integers, representing the numerator and denominator of the number, for example, 15/16, -26/3.
- Floating-point Numbers. Used for approximate mathematical calculations. Symbolics Common Lisp supports two forms:

  - ° *Single-float.* Single-precision floating-point numbers, for example, 1.0e-45.
  - ° *Double-float.* Double-precision floating-point numbers, for example, 5.0d-324.

- Complex Numbers. Used to represent the mathematical concept of that name, for example, #c(4.0 10).

In conventional computer systems, considerations such as number length, base, or internal representation are important and numbers therefore have "computer" properties. In Symbolics Common Lisp, rational numbers are represented as numbers since their representation is not limited by machine word width, but only by total memory limitations. Thus, rational numbers in Symbolics Common Lisp have more familiar mathematical properties.

For internal efficiency, Symbolics Common Lisp also has two primitive types of integers: *fixnums* and *bignums.* Fixnums are a range of integers that the system can represent more efficiently, while bignums are integers outside the range of fixnums. When you compute with integers, the system automatically converts back and forth between fixnums and bignums based solely on the size of the integer. With the exception of some specialized cases, the distinctions between fixnums and bignums are invisible to you, in computing, printing or reading of integers.

The system canonicalizes numbers, that is, it represents them in the lowest form.

Rational canonicalization is the automatic reduction and conversion of ratios to integers, if the denominator evenly divides the numerator.

Integer canonicalization is the automatic internal conversion between fixnums and bignums to represent integers.

Complex canonicalization is the matching of complex number types and the conversion of a complex number to a noncomplex rational number when necessary.

Typically, functions that operate on numeric arguments are *generic*, that is, they work on any number type. Moreover, arithmetic and numeric comparison functions also accept arguments of dissimilar numeric types and *coerce* them to a common type by conversion. When these functions return a number, the coerced type is also the type of the result. Coercion is performed according to specific rules.

Functions are available to let you force specific conversions of numeric data types (for example, convert numbers to floating-point numbers, convert noncomplex to rational numbers).

When comparing numbers, note that although the predicates **eq**, **eql**, **equal**, and **equalp** accept numbers as arguments, they don't always produce the expected results. It is therefore preferable to use = to test numeric equality.

Integer division returns an exact rational number result, that is, it does not truncate the result. (Integer division in Zetalisp truncates the result.)

Operations with numbers include type-checking (**rationalp**), arithmetic, numeric comparison (=), and transcendental functions (**exp**); you can also do bit-wise operations (**logior**, **byte-position**), random number generation, and machine-dependent arithmetic.

Some other terminology associated with numbers:

Radix	An integer that denotes the base in which a rational number prints and is interpreted by the reader. The default radix is 10 (decimal), and the range is from 2 to 36, inclusive. Current radix for printing and reading is controlled by the variables ***print-base***, and ***read-base*** respectively.
Radix specifier	A convention for displaying a rational number with its current radix. For example, `#2r101` is the binary representation of 101. Controlled by the value of the variable ***print-radix***.
Exponent marker	A character that indicates the floating-point format (double, long, single, short) of a floating-point number. Controlled by the value of the variable ***read-default-float-format*** for printing and reading.

1.3 Overview of Symbols

A *symbol* is a Lisp object in the Lisp environment. A symbol has a *print name*, a *value* (or *binding*), a *definition* (or the contents of its *function cell*), a *property list*, and a *package*. It is important to understand that a symbol can be any Lisp object, for example a variable, a function, or a list. It is also important to keep in mind that while we sometimes say that a symbol is the name of some object, a *name* is actually the printed representation of that object. A symbol is the object itself.

Two kinds of symbols should be mentioned explicitly here: Keywords and Variables.

Keywords are implemented as symbols whose home package is the **keyword** package. (See the section "Package Names", page 645.) The only aspects of symbols significant to keywords are name and property list; otherwise, keywords could just as easily be some other data type. (Note that keywords are referred to as enumeration types in some other languages.)

There are three kinds of variables: *special* (or *global*), *local* (or *lexical*), and *instance*. A special variable has dynamic scope: any Lisp expression can access it simply by referring to its name. A local variable has lexical scope: only Lisp expressions lexically contained in the special form that binds the local variable can access it. See the section "Overview of Dynamic and Lexical Scoping", page 57. An instance variable has a different kind of lexical scope: only Lisp expressions lexically contained in methods of the appropriate flavor can access it. Instance variables are explained in another section. See the section "Overview of Flavors", page 47.

1.4 Overview of Lists

This section introduces the concepts of Lisp lists, the components of lists, and other data structures that are composed of lists.

Lists and list-like structures exist to organize data in tabular structures. The simplest such structure is just a collection of items. For example:

scallop
clam
oyster
mussel

The kinds of things a program might do with such a structure are, for example,

- find a given -- first, last, second -- item in the collection/table/list
- see if a given item is included
- add an item to or remove an item from the structure
- copy the structure

or many other possible operations. This collection, which is just a plain list of items, approximately models the mathematical concept of a *set*. Since the need for this kind of structure and these operations is ubiquitous in the type of programming that the Lisp language was designed for, Lisp has an enormous collection of functions for performing these types of operations.

The Cons

The basic data type upon which all tabular structures are based is a record structure called a *cons*. A cons has two components: the head of the cons, which is called the *car*, and the rest, or tail, of the cons, which is called the *cdr*. The basic operations on the cons data type are:

cons and **xcons**	Create a cons with a specified *car* and *cdr*.

consp	Return t if an object is a cons.
car	Return the *car* of the cons.
cdr	Return the *cdr* of the cons.

With the **cons** data type and its associated operations, it is possible to create an unlimited variety of tabular structures. The simplest such structure is the *list.*

Simple Lists

A list is not a primitive Lisp data type; rather, it is a record structure created out of conses. The method by which this construction is done allows the many special list operations to be defined recursively. The key to the construction of a list using conses is the object called **nil**, which is by definition the *empty list.* **nil** is also represented as (). **nil** has its own special data type, **null**, which includes **nil** as its only case.

Having this special object to connote an empty list, it is now easy to define a list in terms of conses:

A list is either **nil** or it is a cons whose tail (*cdr*) is a list.

The list of the above example can thus created by:

```
(cons 'scallop (cons 'clam (cons 'oyster (cons 'mussel
'nil)))) => (SCALLOP CLAM OYSTER MUSSEL)
```

which is equivalent to

```
(list 'scallop 'clam 'oyster 'mussel)
```

Note that the printed form of the list is enclosed within parentheses. This structure could be diagrammed as:

```
 first cons
 car   cdr
  |     |
SCALLOP |
        |
   second cons
     car   cdr
      |     |
    CLAM    |
            |
        third cons
         car   cdr
          |     |
        OYSTER  |
                |
fourth cons
 car   cdr
  |     |
MUSSEL  |
        |
       nil
```

Note that only the heads (*cars*) of the conses of this structure contain the elements of the list. The tail (*cdr*) of each cons contains the rest of the list, except for the last *cdr*, which contains **nil**.

If you study the form of this structure and consider its recursive generation, you can easily see how easy it is to generate recursive functions to search through lists, extract various parts of lists, and the like.

Complex Lists

- Property Lists

At the next level of complexity after the simple kind of list described above, is a table in which each of the items has some property associated with it. For example, a property list for a scallop might be:

> outer-color blue-black
> interior-color mother-of-pearl
> shell thin
> culinary-value high

The kinds of operations that might be performed on a structure like this are adding and removing properties and finding a property, given an item. For these simple operations, a special kind of simple list, called a *property list* is sufficient. A property list is just a list that has an even number of elements that are alternately items and the items' properties. For example, the above list would be represented as

```
(OUTER-COLOR BLUE-BLACK INTERIOR-COLOR MOTHER-OF-PEARL SHELL THIN
CULINARY-VALUE HIGH)
```

The first items of the pairs in the list are called *indicators* and the second members are called *values* or *properties*.

The functions for manipulating property lists are side-effecting operations that have the result of altering the property list itself, rather than of creating a new list.

- Dotted Lists

A cons whose tail (*cdr*) is not the empty list is called a *dotted list*, an unfortunate term since a dotted list is not a true list at all. The "dotted" part of the name stems from the way a dotted list is represented in print with the *car* and *cdr* separated by a dot:

```
(cons 'scallop 'clam) => (SCALLOP . CLAM)
```

Dotted "lists" are the building block for a more complicated structure called an association list.

- Association Lists

Another type of table is one in which each of the items in the table is identified according to some key. For example:

pectinidae	scallop
pelecypoda	clam
ostrea	oyster
mytilus	mussel

The structure used to represent this sort of table is called an association list, or alist. An alist is a list, the elements of which are conses. The *car* of one of these conses is called a *key*, and the *cdr* is called a *datum*. The table above is represented as:

```
((PECTINIDAE . SCALLOP)(PELECYPODA . CLAM)(OSTREA . OYSTER)(MYTILUS .
MUSSEL))
```

The same kinds of operations that can be performed on property lists can be performed on association lists, but because of their more complicated structure, additional operations can be performed on them. You can, for example, search on a key through an alist to find a datum *or* on a datum to find a key. There is a function, **pairlis**, that takes two lists and creates an association list that pairs elements from each.

Association lists can be incrementally updated by adding new entries to the front.

- Trees

Trees are structures composed of one cons and possibly other conses that are associated with that cons, as in these examples:

```
(MYTILUS . MUSSEL)
((PECTINIDAE . SCALLOP)(PELECYPODA . CLAM))
((MYTILUS . MUSSEL)(WHELK PERIWINKLE (FAMILIES . 5) SHELLS)(7 . 4))
```

This last tree could be diagrammed as:

```
      car--------------------------cdr
       |                            |
    car-cdr              car-----------------cdr
     |   |                |                   |
MYTILUS MUSSEL      car--------cdr        car---cdr
                     |          |          |     |
                  WHELK         |       car-cdr nil
                         car--------cdr  |   |
                          |          |   7   4
                     PERIWINKLE      |
                               car--------cdr
                                |          |
                             car-cdr    car-cdr
                              |   |      |   |
                        FAMILIES  5  SHELLS nil
```

• Circular Lists

A circular list is a simple list whose last cons's tail is the first cons of the list. The conses are linked together in a ring with the *cdr* of each cons being the next cons in the ring. This list type is useful especially for those functions that perform a specified operation on all the elements of a list, for example, **mapcar**. Circular lists must be used carefully, however, for they can cause many list functions to get into infinite loops.

• Cdr-Coding

Symbolics Common Lisp uses a special internal representation for conses and lists that effects a substantial reduction in the storage required for these structures. *Cdr-coded* lists require, in the optimum case, only half the space that regular lists use. The disadvantage of cdr-coded lists is that, once they have been altered by operations like **rplacd**, **nconc**, and **nreverse**, access to them can be slowed down considerably.

Cdr-coded lists are created by **list**, **list-in-area**, **make-list**, or **append**.

Normal, that is, not-cdr-coded lists are created by **cons**, **xcons**, or **ncons**, and their in-area variants.

The **copylist** function can be used to convert a normal list into a cdr-coded list.

1.5 Overview of Arrays

1.5.1 Basic Concepts of Arrays

An *array* is a Lisp object that consists of a group of elements. Each array element is a Lisp object. *General arrays* allow the elements to be any type of Lisp object. *Specialized arrays* place constraints on the type of Lisp objects allowed as array elements.

The individual elements of an array are identified by numerical *subscripts*. When accessing an element for reading or writing, you use the subscripts that identify that element. The number of subscripts used to refer to one of the elements of the array is the same as the dimensionality of the array. Thus, in a two-dimensional array, two subscripts are used to refer to an element of the array.

The lowest value for any subscript is 0; the highest value is a property of the array. Each dimension has a *size*, which is the lowest integer that is too great to be used as a subscript. For example, in a one-dimensional array of five elements, the size of the one and only dimension is five, and the acceptable values of the subscript are 0, 1, 2, 3, and 4.

The number of dimensions of an array is called its *dimensionality*, or its *rank*. The dimensionality can be any integer from zero to seven, inclusive. A zero-dimensional array has exactly one element.

The basic functions related to arrays enable you to create arrays (**make-array**), access elements (**aref**), and alter elements (**setf** used with **aref**). Several advanced and more specialized programming practices are also supported: See the section "Advanced Concepts of Arrays", page 22.

A one-dimensional array is known as a *vector*. A *general vector* allows its elements to be any type of Lisp object. *Strings* are vectors that require their elements to be of type **string-char** or **character**. *Bit-vectors* are vectors that require their elements to be of type **bit**.

For more information on the types of arrays: See the section "Data Types and Type Specifiers", page 69.

Zetalisp uses a different terminology for array types. A general array is called a Zetalisp **sys:art-q** array. Zetalisp has many types of specialized arrays, such as **sys:art-fixnum** and **sys:art-boolean**. These types are used by **zl:make-array**, which is supported for compatibility with previous releases. For a complete list of Zetalisp array types: See the section "Zetalisp Array Types", page 183.

1.5.2 Simple Use of Arrays

The following brief example illustrates the syntax of the basic functions for creating arrays, reading and writing their elements, and getting information on arrays.

First, we create and initialize an array that could be used to represent an 8-puzzle game. The first argument represents the array's dimensions; this is a two-dimensional array, with three elements in each dimension. The keyword argument **:initial-contents** is the mechanism for initializing the elements of the array.

```
(setq *8-puzzle*
      (make-array '(3 3)
                  :initial-contents
                  '((3 8 1)
                    (4 5 nil)
                    (2 7 6))))

=>#<ART-Q-3-3 44003776>
```

make-array returns the array. Its printed representation is #<ART-Q-3-3 44003776>.

The next two forms read the elements specified by subscripts (0 2) and (1 2):

```
(aref *8-puzzle* 0 2) => 1
(aref *8-puzzle* 1 2) => NIL
```

To play the first move in the game, we switch the position of the **nil** with any adjoining element. When **setf** is used with **aref** as follows, the element changes to the new value given.

```
(setf (aref *8-puzzle* 0 2) nil) => NIL
(setf (aref *8-puzzle* 1 2) 1) => 1
```

Instead of continuing with the game, we request information on the ***8-puzzle*** array: why not?

- What is the rank of the array, or how many dimensions does it have?

  ```
  (array-rank *8-puzzle*) => 2
  ```

  The array has 2 dimensions, or a rank of 2.

- What are the dimensions of the array?

  ```
  (array-dimensions *8-puzzle*) => (3 3)
  ```

  The elements of the returned list **(3 3)** are the dimensions of the array.

- What is the type of the elements in the array?

  ```
  (array-element-type *8-puzzle*) => T
  ```

  The returned value, **t**, indicates that the array elements can be of any type.

1.5.3 Advanced Concepts of Arrays

This section introduces some of the advanced topics of arrays as well as terminology associated with those topics.

Array leader	Typically the elements of an array are a homogeneous set of objects. Sometimes it is desirable to store a few nonhomogeneous pieces of data attached to the array. You can use an *array leader* to do this. An array leader is similar to a general one-dimensional array that is attached to the main array. You can create a leader using the **:leader-length** or **:leader-list** option for **make-array**, and examine and store elements in the array leader using numeric subscripts. Alternatively, you can construct the leader using the **:array-leader** option for **defstruct**, and then use automatically generated constructor functions to access the slots of the leader.
Fill pointer	By convention, element zero of the array leader of an array is used to hold the number of elements in the array that are "active" in some sense. When the zeroth element is used this way, it is called a *fill pointer*. Many array-processing functions recognize the fill pointer. For instance, if a string has seven elements, but its fill pointer contains the value 5, then only elements zero through four of the string are considered to be "active". This means that the string's printed representation is five characters long, string-searching functions stop after the fifth element, and so on.
Displaced array	Normally, an array is represented as a small amount of header information followed by the contents of the array. However, sometimes it is desirable to have the header information removed from the array's contents. A displaced array is such an array. You can create one with the **:displaced-to** option to **make-array**.
Indirect array	This is an array whose contents are defined to be the contents of another array. You can create one by giving an array as the value of the **:displaced-to** option to **make-array**.
Index offset	Both indirect and displaced arrays can be created in such a way that when an element is referenced or stored, a constant number is added to the subscript given. This number is called the index offset, and it is specified by giving an integer as the value of the **:displaced-index-offset** option to **make-array**.
Raster	This is a two-dimensional array that is conceptually a rectangle

	of bits, pixels, or display items. A variety of raster operations is available.
Plane	This is an array whose bounds, in each dimension, are plus-infinity and minus-infinity. All integers are valid as subscripts. A variety of plane operations is available.
Array register	When performance is especially important, you can use the array register feature to optimize your code.
Adjusting an array	You can adjust an existing array to give it a new dimensionality. To ensure that an array will be adjustable after it is created, use the **:adjustable** option to **make-array**.
Array storage	In all Lisp dialects supported by Genera, arrays are stored in memory in row-major order. This is an implementation detail that does not concern most programmers. However, if you use some of the advanced array practices, such as displaced arrays or adjusting the array size dynamically, you need to understand how arrays are stored in memory.

1.6 Overview of Sequences

A *sequence* is a data type that contains an ordered set of elements. It subsumes the types lists and vectors (one-dimensional arrays).

Symbolics Common Lisp provides a range of general sequence functions that operate on both lists or vectors. These functions perform basic operations on sequences, irrespective of their underlying representation. The advantage of using a sequence operation, rather than one specifically for lists or vectors, is that you need not know how the sequence has been implemented. It makes sense to reverse a sequence or extract a range of sequence elements, whether the sequence is implemented as a vector or a list.

The principal operations on sequences fall into the following categories:

- Construction and access
- Predicates
- Mapping
- Modification
- Searching
- Sorting and merging

Argument keywords extend the power of the sequence functions. For example, the keywords **:test**, **:test-not**, and **:key** allow you to set up arbitrarily complex tests for customizing the operation of the sequence functions. See the section "Testing Elements of a Sequence", page 190.

1.7 Overview of Characters

A *character* is a type of Lisp object. A character object is used to represent letters of the alphabet and numbers, among other things. Characters are the building blocks of strings; a string is a one-dimensional array of characters.

The reader recognizes characters by the #\ prefix followed by the character. For example: #\A is read as the character A; #\1 is read as the character 1. Non-printing characters have names; the reader recognizes them by the #\ prefix followed by a name, such as #\Space.

Each character object has the following attributes: the character code, the character set, the character bits, and the character style.

A *character set* is a group of related characters. All characters in a character set are recognized as belonging together, even if they are different sizes or styles.

Genera supports three character sets: the Symbolics standard character set, the mouse character set, and the arrow character set. Characters that are in character sets other than the Symbolics character set are represented by the #\ prefix followed by the name of the character set, a colon, and the name of the character. For example:

```
#\mouse:scissors
#\arrow:eye
```

Two characters of different character sets can never be **char-equal**.

The *character code* is the attribute of a character that identifies the particular character in the same way that ASCII codes represent particular characters. Two characters in different sets never have the same code. For example, the Symbolics standard character set a and the Greek character set α have different character codes. (Note that Genera does not support a Greek character set.)

The *character bits* are an attribute of characters. The bits represent the hyper, super, meta, and control keys; they make it possible to distinguish between the character "A" and the character "control A", for example.

Characters that have bits set are read by the reader in the same way that other characters are read: the #\ prefix is followed by the character's name. For example, #\control-A or #\c-A is read as the character "control A". Other examples are: #\c-m-Return, #\hyper-Space, #\meta-B.

A *character style* is a combination of three characteristics that describe how a character appears. These characteristics are the *family*, *face*, and *size*.

Family	Characters of the same family have a typographic integrity, so that all characters of the same family resemble one another. Examples: SWISS, DUTCH, and FIX.
Face	A modification of the family, such as BOLD or ITALIC.
Size	The size of the character, such as NORMAL or VERY-SMALL.

The character style is the grouping of the family, face, and size fields. A character style is often represented by the convention:

family.face.size

An example of a fully specified character style is:

```
SWISS.ITALIC.LARGE
```

Each element of the character style can be specified or left unspecified. A family, face, or size of NIL means to use the default value. Most characters have the following character style:

```
NIL.NIL.NIL
```

Characters of style NIL.NIL.NIL are displayed in the default character style established for the current output device.

Genera distinguishes between *thin* and *fat* characters:

Thin character	A character whose character style is NIL.NIL.NIL and whose bits are all zero. Thin characters are of type **string-char**. For example: #\A
Fat character	A character that has a character style other than NIL.NIL.NIL or whose modifier bits are set to something other than zero. Fat characters are of type **character**. For example: #\c-A

describe is useful for getting information about a character. It responds with the character's bits, style, code, and character set; it returns the character itself.

The following example shows the result of describing a thin character representing the letter A.

```
(describe #\A) =>
#\A is a character with bits #b0, style NIL.NIL.NIL, and code 65
This is offset 65 in character set #<STANDARD-CHARACTER-SET 204000540>
#\A
```

The following example shows the result of describing a fat character that represents the letter A. This character has the Meta bit set and has the style NIL.ROMAN.NIL. However, the character code of this fat character is the same as the character code of the thin character representing the letter A.

```
(describe (make-character #\A :bits char-meta-bit
                              :style '(nil :roman nil))) =>
#\m-sh-A is a character with bits #b10, style NIL.ROMAN.NIL, and code 65
This is offset 65 in character set #<STANDARD-CHARACTER-SET 204000540>
#\m-sh-A
```

Character styles are device independent. When you want to display a character on a specific device (such as the black and white console, or the LGP2 printer), a specific *font* must be chosen to represent the character. The font is chosen depending on: the character code, the character set, the character style, and the device type. The system has a set of predefined mappings between character sets, character styles, device types and specific fonts.

Common Lisp has a font field instead of a character style field. As implemented in SCL, characters have no font field and the **char-font-limit** is **1**. This is in compliance with Common Lisp.

In Symbolics documentation the word font is used in two contexts: to describe a font that is specific to a device for representing characters; to refer to the font of a character as implemented in releases of Symbolics software prior to Genera 7.0.

Mouse characters and the functions that manipulate them are described elsewhere: See the section "Mouse Characters" in *Programming the User Interface, Volume B.*

1.8 Overview of Strings

The Lisp object, *string*, is a specialized type of *vector*, or one-dimensional array, whose elements are characters.

Common Lisp defines a string as a vector whose elements are characters of type **string-char**. Symbolics Common Lisp extends this definition by recognizing an additional string type, namely a vector whose elements are of type **character**. Strings of type **string-char** are called *thin* strings; they are made up of *thin* characters. Strings of type **character** are called *fat* strings; they are made up of *fat* characters.

Thin string	An array whose elements are thin characters (standard characters of type **string-char** with no character style or modifier bits attributes). For example, "any string". The predicate **string-char-p** tests for thin characters.
Fat string	An array whose elements are fat characters of type **character**, with fields holding information about character style and modifier bits. For example, "**any string**". The predicate **string-fat-p** tests strings for fatness.

Characters and their attributes are discussed elsewhere in this Overview: See the section "Overview of Characters", page 24.

The function **stringp** lets you test any Lisp object to determine if it is a string.

Zetalisp uses a different terminology for string types. A *thin* string is called **sys:art-string**, and a *fat* string is called **sys:art-fat-string**.

Common Lisp also distinguishes between the type *string* and a subtype of it called *simple-string*. A *simple-string* is a *simple-array*, that is, an array that has no fill pointer, is not adjustable after creation, and whose contents are not displaced (shared with another array). A *string* is an array that can have a fill pointer, can be adjusted after creation, and can be displaced. The types of arrays are discussed elsewhere in this Overview: See the section "Advanced Concepts of Arrays", page 22.

The predicates **string-p** and **simple-string-p** test if an object is a *string* or a *simple-string*. The distinction between strings and simple strings is not especially important in Symbolics Common Lisp.

The individual elements of string arrays are identified by numeric subscripts; when accessing portions of a string for reading or writing you use the subscript to identify the elements. The subscript count always begins at zero. In many cases string operations also return an integer that is an index into the string array (as for example to indicate the position of a character found in a string search).

As vectors, strings constitute a subtype of the type **sequences**. Hence, many string operations can use general purpose array or sequence functions; a large number of string-specific functions are also available.

The basic functions relating to strings let you create strings (**make-string**, or **make-array**), access a single string element (**char**, or **aref**), modify strings or portions of them (**setf** used with **char** or **aref**), and get information about string size (**string-length**). Other typical string operations for which a variety of functions are provided include comparing two strings, altering string case, removing portions of a string, combining strings, and searching a string for a character or a string of characters.

String comparisons and searches examine every individual element of the string. The *case-sensitivity* of the comparison determines which attributes of a character are respected or ignored.

A *case-sensitive* operation takes into account every single attribute of the characters compared, whereas a *case-insensitive* operation ignores the attributes specifying character *style* and character *case*. Both case-sensitive and case-insensitive operations compare attribute fields such as character code and modifier bits.

For example:

```
(string= "sail" "SAIL") => NIL
; case-sensitive comparison fails

(string-equal "sail" "SAIL") => T
; case-insensitive comparison succeeds
```

The case-sensitive string comparison functions are distinguished by their use of algebraic comparison symbols as suffixes (for example, **string=**); the case-insensitive string comparison functions have alphabetic symbols as suffixes (for example, **string-equal**, **string-lessp**).

The case-sensitive string search functions often use the suffix **-exact** (for example **string-search-exact-char**); the case-insensitive string search functions omit this suffix (for example, **string-search-char**).

Many string functions can be *destructive* or *non-destructive* with respect to their argument(s). Functions beginning with the character "n" modify their argument so that its original form is destroyed (for example, **string-nreverse** which reverses the characters of its argument and does not preserve it). Such destructive functions have a non-destructive counterpart which preserves the original argument and returns a modified copy of it (for example **string-reverse**).

Examples:

```
; non-destructive lowercasing operation preserves
; the original argument
(setq original "THREE BLIND MICE")  => "THREE BLIND MICE"
(string-downcase original) => "three blind mice"
original => "THREE BLIND MICE"

; destructive lowercasing - original argument is lost
(setq original "THREE BLIND MIÇE")  => "THREE BLIND MICE"
(nstring-downcase original)  => "three blind mice"
original  => "three blind mice"
```

Most string operations use *keyword* arguments to help you customize the extent and the direction of the operation. Keyword arguments are prefixed by a : character; the most important are **:start**, **:end**, and **:from-end**.

:start and **:end** must be non-negative integer indices into the string array, and **:start** must be smaller than or equal to **:end**. These keywords operate only on the "active" portion of the string, that is, the portion below the limit specified by the fill pointer, if there is one. **:start** indicates the start position for the operation within the string. It defaults to zero (the start of the string). **:end** indicates the position of the first element in the string *beyond* the end of the operation. It defaults to **nil** (the length of the string). If both **:start** and **:end** are omitted, the entire string is processed by default.

For example:

```
; to capitalize the last four characters in "applejack"
(string-upcase "applejack" :start 5)   => "appleJACK"

; to reverse the middle three characters of "doodle"
(string-reverse "doodle" :start 1 :end 4)  => "ddoole"
```

If two strings are involved, the keyword arguments **:start1**, **:end1**, **:start2**, and **:end2** are used to specify substrings for each separate string argument.

For example:

```
; to compare the first three characters of two strings
(string= "apple" "applejack" :end1 3 :end2 3) => T
```

For operations such as searches, it can be useful to specify the direction in which the string is conceptually processed. This is controlled by the keyword argument **:from-end**.

Where this keyword is present in the argument list, the function normally processes the string in the forward direction, but if a non-**nil** value is specified for **:from-end**, processing starts from the reverse direction. Regardless of the direction of processing, the count indicating the position of the item found always starts from the beginning of the string.

For example:

```
(string-search-exact #\e "heavenly") => 1
       ; normal search, returns the position of the
       ; first (leftmost) occurrence
       ; of the character "e"

(string-search-exact #\e "heavenly" :from-end t) => 4
       ; reverse search, returns the position of the last
       ; (rightmost) occurrence of the character "e"
       ; counting from the beginning of the string
```

1.9 Cells and Locatives

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 a **sys:art-q** array of *n* elements has *n* cells. (Numeric arrays do not have cells in this sense.)

A *locative* is a type of Lisp object used as a *pointer* to a single memory cell anywhere in the system; it lets you refer to a cell, so that you can examine or alter its contents. Locatives are inherently a more "low-level" construct than most Lisp objects; they require some knowledge of the nature of the Lisp implementation. Most programmers never need them.

Here is a list of functions that create locatives to cells:

zl:aloc
zl:ap-leader
zl:car-location
zl:value-cell-location
sys:function-cell-location

Each function is documented with the kind of object to which it creates a pointer.

The macro **locf** can be used to convert a form that accesses a cell to one that creates a locative pointer to that cell.

For example:

```
(locf (fsymeval x)) ==> (sys:function-cell-location x)
```

locf is very convenient because it saves the writer and reader of a program from having to remember the names of all the functions that create locatives. See the section "Generalized Variables", page 503.

The contents of a cell can be accessed by **location-contents** and updated by **(setf (location-contents ...))**.

Access to and modification of the contents of locatives is currently implemented by the system using the operations **cdr** and **rplacd**. Therefore, these instructions may appear in the disassembly of compiled programs which operate on locatives. Also, you may sometimes see these functions used to manipulate locatives in old code. This usage is obsolete and should not be employed in new software.

1.9.1 Table of Functions That Operate on Locatives

location-boundp *location*	Tests if the cell at location is bound to a value
location-contents *locative*	Returns the contents of the cell pointed to by *locative*
location-makunbound *loc*	Causes the cell at *loc* to become unbound
locativep *object*	Tests if *object* is a locative
locativep *x*	Tests if *x* is a locative
locf *access-form*	Converts *access-form* to a new form that will create a locative pointer to that cell

2. Functions, Predicates, Special Forms and Macros

2.1 Overview of Functions

Functions are the basic building blocks of Lisp programs. A *function* is a Lisp object that, when applied to arguments, performs some action and returns a value. You can manipulate functions in the same ways you manipulate other Lisp objects; you can pass them as arguments, return them as values, and make other Lisp objects refer to them.

There are four kinds of functions, classified by how they work:

- *Interpreted* functions, which are defined with **defun**, represented as list structures, and interpreted by the Lisp evaluator.

- *Compiled* functions, which are defined by compiling forms from a file or an editor buffer or by loading a binary file, are represented by a special Lisp data type, and are executed directly by the machine.

- Various types of Lisp objects that can be applied to arguments, but when applied, call another function and apply it instead. These include symbols, dynamic and lexical closures, and instances.

- Various types of Lisp objects that, when used as functions, do something special related to the specific data type. These include arrays and stack groups.

Lisp has several functions known as *function-defining special forms*, which are used in programs to define functions. For example, **defun** is a common function-defining special form. Function-defining special forms typically take as arguments a function spec (see below) and a description of the function to be made.

Function-defining special forms include **defun**, **defsubst**, **macro**, **defselect**, **deff**, and **def**.

A general programming-style rule of thumb: Anything that is used at top level (not inside a function) and starts with **def** should be a function-defining special form so that the editor can find it in your source file and show it to you whenever you ask for a definition.

For more information on function-defining special forms: See the section "Function-Defining Special Forms", page 258.

The name of a function is usually a symbol, but does not have to be a symbol. A function can be represented by a *function spec*, which serves to name a function and specifies a place to find and remember a function. *Spec* is short for *specification.*

Function specs are not functions. You cannot apply a function spec to arguments. You use function specs when you want to do something to the function, such as define it, look at its definition, or compile it. Both function specs and functions can be defined. To define a function spec means to make that function spec remember a given function – a task accomplished by the **fdefine** function. To define a function means to create a new function and define a given function spec as that new function – a task accomplished by the **defun** special-form. Several other special forms, such as **defmethod** and **defselect**, also define functions.

A function spec's definition generally consists of a *basic definition* surrounded by *encapsulations.* The basic definition is what **defun** creates. See the section "How Programs Manipulate Definitions", page 261. The encapsulation is composed of function-altering functions, such as **trace** and **advise**. See the section "Encapsulations", page 263. When the function is called, the function's definition plus the alterations are executed.

For more information on function specs: See the section "Function Specs", page 251.

There are several operations a user would typically want to perform on functions. These operations are:

- Print out the definition of the function spec with indentation. (This works only with interpreted functions.)
- Find out about a function by looking at its documentation and its arguments.
- Look at the function's debugging information.
- Trace the calling history and customize the definition of a function while debugging.
- Examine the compiled code, if the function is compiled.

For more information on these operations: See the section "Operations the User Can Perform on Functions", page 255.

A Lisp *definition* is a Lisp expression that appears in a source program file and has a name to which a user can refer. Two definitions with the same name and different types can exist simultaneously, but two definitions with the same name and the same type redefine each other when evaluated. There are four basic types of definitions:

- functions

- variables
- flavors
- structures

Many types of Lisp special forms, such as **defun** and **defvar**, can define these four types of definitions. For more information about definitions: See the section "How Programs Manipulate Definitions", page 261.

A Lisp *declaration* is an optional Lisp expression that provides the Lisp system with information about your program, for example, documentation. Many Lisp forms, such as **defun**, have declarative aspects. See the section "Declarations", page 260.

A *dynamic closure* is a Lisp functional object for implementing certain advanced access and control structures. Closures give you more explicit control over the environment, by allowing you to save the environment created by the entering of a dynamic contour, and then use that environment elsewhere, even after the contour has been exited. There are several functions that manipulate dynamic closures, for example, **closure**.

For more information on dynamic closures: See the section "Dynamic Closures", page 265.

2.2 Overview of Predicates

A *predicate* is a function that tests for some condition involving its arguments and returns some non-**nil** value if the condition is true, or the symbol **nil** if it is not true. Many predicates return the symbol **t**, instead of another non-**nil** value, if the condition is true.

Predicate names usually end in the letter "p". The way the "p" is added to the end of the predicate depends on whether or not there is an existing hyphen in the name. For example, the predicate that tests for integers is **integerp**, while the predicate that checks for compiled functions is **compiled-function-p**.

Predicates fall into several logical categories. These include: type-checking predicates, which test an object for membership in a particular data type such as numbers, arrays, and so on; property-checking predicates, which determine whether an object has certain properties (such as whether a number is odd or even); comparison predicates, which compare objects of the same type; and a few others.

For a complete list of predicates: See the section "Predicates", page 271. A full description of each predicate is available in the dictionary of Lisp functions.

2.3 Overview of Macros

The macro facility allows the user to define arbitrary functions that convert certain Lisp forms into different forms *before* evaluating or compiling them.

This is done at the expression level, not at the character-string level, as in most other languages. Macros are important in the writing of good code: they make it possible to write code that is clear and elegant at the user level, but that is converted to a more complex or more efficient internal form for execution.

When **eval** is given a list whose *car* is a symbol, it looks for local definitions of that symbol; if that fails, it looks for a global definition. If the definition is a macro, it contains an *expander* function. **eval** applies the expander function to two arguments: the form that **eval** is trying to evaluate, and an object representing the lexical environment. The expander function returns a new form. This is the *expansion* of the macro call. **eval** evaluates this expansion in lieu of the original form.

An example of a macro expansion would be as follows :

```
(macroexpand '(return x))
 => (RETURN-FROM NIL
      X) and T
```

Macros are used for a variety of purposes, the most common being extensions of the Lisp language. Note that macros are not functions, and cannot be applied to arguments.

The **defmacro** construct provides a convenient way to define new macros, and the *backquote facility* helps to increase their readability. Backquote (‘) is a *reader macro* that generates lists. In simple cases, the backquote is just like the regular single quote macro: it creates a form that when evaluated produces the form following the backquote. For example:

```
`(1 2 3) => (1 2 3)
'(1 2 3) => (1 2 3)
```

If you include a comma (,) inside the form following a backquote, that form gets evaluated even though it is inside the backquote. For example:

```
(setq b 1)
`(a b c) => (A B C)
`(a ,b c) => (A 1 C)
```

In other words backquote quotes everything except things preceded by a comma; those things get evaluated.

If an at-sign character follows the comma (,@), it has a special meaning. This construct should be followed by a form whose value is a list; then each of the elements of the list is appended to the list being created by the backquote. For example:

```
(setq a '(x y z))
`(1 ,a 2)  => (1 (X Y Z) 2)
`(1 ,@a 2) => (1 X Y Z 2)
```

Here is a simple macro definition using the backquote facility:

```
(defmacro onep (num)
   `(zerop ,(- 1 num))) => ONEP
(onep 1) => T
(onep 0) => NIL
```

Inline functions are somewhat similar to macros. An inline function executes like a function; if it is called by another function that is being compiled, the inline function's definition is substituted into the code being expanded. In this respect, an inline function is like a macro. If something can be implemented as either a macro or an inline function, it is generally better to make it an inline function.

2.4 Special Forms and Built-in Macros

In order to define the terms "special form" and "macro" it is necessary first to review some basic concepts.

The *form* is the standard evaluation unit in Lisp. It is a data object that is meant to be *evaluated* as a program to produce one or more *values* (which are also data objects). See the section "Introduction to Evaluation", page 493. There are three categories of forms:

- self-evaluating forms, such as numbers, characters, strings, and bit-vectors
- symbols, which stand for variables
- lists

The evaluator, when applied to a list, performs the operation specified by the first element of the list, in order to produce a value to return. The first element of the list is referred to as an *operator*. There are two categories of operators:

- functions
- special operators

Functions are explained at length in their own chapter. See the section "Functions", page 251.

There are two kinds of special operators:

- special forms
- macros

A *special form* is a special operator that is "built in" to the Lisp language; that is, this type of special operator is contained within the compiler and interpreter. (Sometimes special forms are referred to as *primitive special operators.* This latter term more accurately expresses the concept, since a special form is not really a "form" at all. The term "special form" is the one that has been in use in the Lisp literature heretofore, so the current documentation retains it for the sake of consistency.)

Most special forms are either control constructs (for example, **case**, **do**, **loop**) or environment constructs (for example, **let**, **defconstant**). Evaluation of some special forms calls for a nonlocal exit rather than returning a value. An example is **throw**. There is no general syntax for a special form; each special form has its own syntax.

A built-in macro is also defined and available within the language, but unlike special forms, macros can also be defined by the user.

A *macro call* is a list whose first element is the name of a macro. Each macro has its own expander function. When a macro call is made, the expander function computes a new form that is to be evaluated in place of the original form. The resulting value is returned as the value of the original form. See the section "Introduction to Macros", page 285.

The definition of a special form can not be moved from one symbol to another, while the definition of macro, or a function, can. Whether a particular special operator is a special form or a macro is implementation dependent. An implementation is free to implement any special form as a macro and vice versa. The user can define new functions and macros, but the set of special forms is fixed by the implementation.

The following tables list the special forms and built-in macros in Symbolics Common Lisp, Common Lisp, and Zetalisp, categorized according to the chapter in which they are presented.

2.4.1 Table of Special Forms

The following is a list of different chapters in Reference Guide to Symbolics Common Lisp in which Special Forms appear:

- Numbers
- Functions
- Macros
- Flavors

- Evaluation
- Scoping
- Flow of Control
- Conditions
- Packages

2.4.1.1 Special Forms Appearing in the Chapter Numbers

Function name	*Appears in the table*
zl:signp	"Numeric Property-checking Predicates"

2.4.1.2 Special Forms Appearing in the Chapter Functions

declare
def
deff
defselect
defun
zl:let-closed
zl:local-declare

2.4.1.3 Special Forms Appearing in the Chapter Macros

Function name	*Appears in the section*
deffunction	"Lambda Macros"
define-symbol-macro	"Symbol Macros"
deflambda-macro	"Lambda Macros"
zl:deflambda-macro-displace	"Lambda Macros"
defsubst	"Inline Functions"
lambda-macro	"Lambda Macros"
macro	"Introduction to Macros"

2.4.1.4 Special Forms Appearing in the Chapter Flavors

Function name	*Appears in the table*
defgeneric	"Summary of Flavor Functions and Variables"
define-method-combination	"Summary of Flavor Functions and Variables"
define-simple-method-combination	"Summary of Flavor Functions and Variables"
defmacro-in-flavor	"Summary of Flavor Functions and Variables"

defselect-method	"Summary of Flavor Functions and Variables"
defsubst-in-flavor	"Summary of Flavor Functions and Variables"
defun-method	"Summary of Flavor Functions and Variables"
defun-in-flavor	"Summary of Flavor Functions and Variables"
defwhopper	"Summary of Flavor Functions and Variables"
undefun-method	"Summary of Flavor Functions and Variables"

2.4.1.5 Special Forms Appearing in the Chapter Evaluation

zl:comment
compiler-let
zl:defconst
defconst
defconstant
defparameter
defsetf
defvar
zl:desetq
zl:destructuring-bind
destructuring-bind
zl:dlet
zl:dlet*
function
lambda
let
let*
let-globally
let-if
letf
letf*
zl:multiple-value
multiple-value-bind
multiple-value-call
multiple-value-list
multiple-value-prog1
prog1
prog2
progn
progv
zl:progv
progw
psetf
psetq

zl:psetq
quote
setq

2.4.1.6 Special Forms Appearing in the Chapter Scoping

flet
labels
macrolet

2.4.1.7 Special Forms Appearing in the Chapter Flow of Control

Function name	*Appears in the table*
zl:*catch	"Nonlocal Exit Functions"
and	"Conditional Functions"
block	"Blocks and Exits Functions and Variables"
case	"Conditional Functions"
zl:caseq	"Conditional Functions"
catch	"Nonlocal Exit Functions"
ccase	"Conditional Functions"
cond	"Conditional Functions"
cond-every	"Conditional Functions"
ctypecase	"Conditional Functions"
zl:dispatch	"Conditional Functions"
do	"Iteration Functions"
do*	"Iteration Functions"
zl:do*-named	"Iteration Functions"
zl:do-named	"Iteration Functions"
dolist	"Iteration Functions"
zl:dolist	"Iteration Functions"
dotimes	"Iteration Functions"
zl:dotimes	"Iteration Functions"
ecase	"Conditional Functions"
etypecase	"Conditional Functions"
go	"Transfer of Control Functions"
if	"Conditional Functions"
zl:keyword-extract	"Iteration Functions"
or	"Conditional Functions"
prog	"Iteration Functions"
prog*	"Iteration Functions"
return	"Blocks and Exits Functions and Variables"
return-from	"Blocks and Exits Functions and Variables"

select	"Conditional Functions"
selector	"Conditional Functions"
zl:selectq	"Conditional Functions"
selectq-every	"Conditional Functions"
tagbody	"Transfer of Control Functions"
throw	"Nonlocal Exit Functions"
typecase	"Conditional Functions"
zl:typecase	"Conditional Functions"
unwind-protect	"Nonlocal Exit Functions"
unwind-protect-case	"Nonlocal Exit Functions"

2.4.1.8 Special Forms Appearing in the Chapter Conditions

Function name	*Appears in the table*
argument-typecase	"Condition-Checking and Signalling Functions and Variables"
catch-error-restart	"Restart Functions"
catch-error-restart-if	"Restart Functions"
cerror	"Condition-Checking and Signalling Functions and Variables"
condition-bind	"Basic Forms for Bound Handlers"
condition-bind-default	"Basic Forms for Default Handlers"
condition-bind-default-if	"Basic Forms for Default Handlers"
condition-bind-if	"Basic Forms for Bound Handlers"
condition-call	"Basic Forms for Bound Handlers"
condition-call-if	"Basic Forms for Bound Handlers"
condition-case	"Basic Forms for Bound Handlers"
condition-case-if	"Basic Forms for Bound Handlers"
define-global-handler	"Basic Forms for Global Handlers"
undefine-global-handler	"Basic Forms for Global Handlers"
error-restart	"Restart Functions"
error-restart-loop	"Restart Functions"
ignore-errors	"Basic Forms for Bound Handlers"
signal-proceed-case	"Protocol for Proceeding"

2.4.1.9 Special Forms Appearing in the Chapter Packages

zl:defpackage
do-all-symbols
do-external-symbols
do-local-symbols
do-symbols

zl:package-declare
variable-boundp
zl:variable-location
variable-makunbound

2.4.2 Table of Macros

The following is a list of different chapters in Reference Guide to Symbolics Common Lisp in which Macros appear:

- Lists
- Arrays
- Functions
- Macros
- Structure Macros
- Flavors
- Evaluation
- Flow of Control
- Conditions
- Packages

2.4.2.1 Macros Appearing in the Chapter Lists

pushnew	See the section "Functions for Constructing Lists and Conses", page 144.

2.4.2.2 Macros Appearing in the Chapter Arrays

array

2.4.2.3 Macros Appearing in the Chapter Functions

defunp
@define
si:encapsulate
let-and-make-dynamic-closure

2.4.2.4 Macros Appearing in the Chapter Macros

Function name	*Appears in the section*
defmacro	"Aids for Defining Macros"
once-only	"Multiple and Out-of-order Evaluation"

defmacro-displace	"Displacing Macros"

2.4.2.5 Macros Appearing in the Chapter Structure Macros

defstruct-define-type
defstruct

2.4.2.6 Macros Appearing in the Chapter Flavors

Function name	*Appears in the table*
compile-flavor-methods	"Summary of Flavor Functions and Variables"
define-simple-method-combination	"Summary of Flavor Functions and Variables"
defwrapper	"Summary of Flavor Functions and Variables"
defwhopper-subst	"Summary of Flavor Functions and Variables"
defflavor	"Summary of Flavor Functions and Variables"
defmethod	"Summary of Flavor Functions and Variables"

2.4.2.7 Macros Appearing in the Chapter Evaluation

setf
locf
deflocf
incf
decf
swapf
push
push-in-area
pop

2.4.2.8 Macros Appearing in the Chapter Flow of Control

Function name	*Appears in the section*
unwind-protect-case	"Nonlocal Exit Functions"
loop	"Iteration Paths"
loop-finish	"End Tests For **loop**"
define-loop-macro	"Introduction To **loop**"
define-loop-sequence-path	"Iteration Paths"
define-loop-path	"Iteration Paths"
when	"**loop** Conditionalizing Keywords"
unless	"**loop** Conditionalizing Keywords"

2.4.2.9 Macros Appearing in the Chapter Conditions

Function name	*Appears in the table*
assert	"Condition-Checking and Signalling Functions and Variables"
catch-error	"Condition-Checking and Signalling Functions and Variables"
check-arg	"Condition-Checking and Signalling Functions and Variables"
check-arg-type	"Condition-Checking and Signalling Functions and Variables"
dbg:with-erring-frame	"Functions for Examining Stack Frames"

2.4.2.10 Macro Appearing in the Chapter Packages

pkg-bind

3. More Complex Constructs: Structures, Flavors, Table Management

3.1 Overview of Structure Macros

Symbolics Common Lisp offers a variety of built-in data types, such as symbols, lists, and arrays. You can use Lisp functions to create a new symbol, set the value of the symbol, read its value, and alter its value. The same functionality is available for lists and arrays.

The structure macro facility enables you to extend Lisp's data types by defining new types of data structures. Once you have defined a new type of data structure, you can create new structures of that type, and then read and set the values of their elements.

The newly defined data structure is a convenient, concise, and high-level way to represent an *object.* For example, if your program simulates an ocean environment, you might need to represent boats. You can use structure macros to define a high-level representation of boats. The elements of the data structure are called *slots.* Further on, we define a sample boat structure that has slots for the boat's x-position, y-position, x-velocity, and y-velocity.

To define new structures, you use **defstruct** or **zl:defstruct**. These macros provide a similar functionality. **defstruct** adheres to the Common Lisp standard, with several extensions derived from useful features of **zl:defstruct**. **zl:defstruct** is supported for compatibility with previous releases.

In brief, the structure macro facility gives you the following features:

- Ability to define new aggregate data structures with named slots.

  ```
  (defstruct boat
    x-position
    y-position
    x-velocity
    y-velocity)
  ```

- Constructor functions (generated automatically) for making objects of the newly-defined type of structure.

  ```
  (setq boat-1 (make-boat))
  ```

- Slot-initialization capabilities, including a way to initialize slot values when constructing new objects, and to specify default slot values in the **defstruct** form.

```
(setq boat-2 (make-boat :x-position 12
                        :y-position 73
                        :x-velocity 0
                        :y-velocity 25))
```

- Accessor functions (generated automatically) for reading the value of a slot.

```
(boat-x-position boat-2)
```

- Alterant macros (generated automatically) for changing the value of a slot.

```
(setf (boat-x-position boat-2) 12.5)
```

By creating a new high-level data structure to represent the objects of a program, you gain several advantages over using lower-level data structures, such as lists or arrays. The program should be more readable and understandable.

For example, if you represent a boat with lists or arrays, it would not be obvious when reading a program that an expression such as **(fifth boat-1)** or **(aref boat-2 4)** means "the y component of the boat's velocity".

The main purpose of using **defstruct** to define new structures is to increase the clarity of a program that deals in some kind of objects. The clarity is a result of named slots and automatically-generated constructor, accessor, and alterant macros.

defstruct offers other features, such as the ability to control the internal representation of the structure. You can use the **:type** option to indicate that the structure should be implemented as a list, an array, a named-array, and so on.

You can also create new structures that inherit slots from another structure. For example, you might define a structure to represent a person. You might then define structures to represent astronauts, which could include the slots of the person structure.

defstruct structures are useful and appropriate for many application programs. Flavors is an alternate method of writing programs that need to represent objects. Flavors offers greater flexibility in program development and several programming practices that are not available with **defstruct** structures.

For related information:

See the section "Structure Macros", page 319. See the section "Differences Between **defstruct** And **zl:defstruct**", page 334. See the section "Overview of Flavors", page 47. See the section "Comparing **defstruct** Structures and Flavors", page 54.

3.2 Overview of Flavors

Flavors is the part of Symbolics Common Lisp that supports object-oriented programming. Flavors is a powerful and flexible tool for programming in a modular style.

The basic concepts of Flavors are simple to understand and it is easy to begin experimenting with Flavors. On the other hand, Flavors is a complex system that offers many advanced options and programming practices. These advanced topics are not presented here, but are covered in the reference documentation: See the section "Flavors", page 353.

3.2.1 Concepts of Flavors

It is often convenient to organize programs around *objects*, which model real-world things. Each object has some *state*, and a set of operations that can be performed on it. Object-oriented programming is a technique for organizing very large programs. This technique makes it practical to manage programs that would otherwise be impossibly complex.

An object-oriented program consists of a set of objects and a set of operations on those objects. The design of such a program consists of three major tasks:

- Choosing the kinds of objects to provide in the program.
- Defining the characteristics of each kind of object.
- Determining what operations can be performed on each kind of object.

Using Flavors terminology, an object-oriented program is built around:

Flavors	Each kind of object is implemented as a *flavor*. A flavor is a template for objects. In other words, a flavor is an abstraction of the characteristics that all objects of this flavor have in common.
Instances of a flavor	Each object is implemented as an *instance* of a flavor. In fact, the term *object* is used interchangeably with *instance*.
Instance variables	Each flavor specifies a set of state variables for objects of that flavor. These are called *instance variables*.
Generic functions	The operations that are performed on objects are known as *generic functions*.
Methods	The code that performs a generic function on instances of a

certain flavor is called a *method*. Typically, one generic function has several methods defined for it.

Often a flavor is defined by combining several other flavors, called its *components*. The new flavor inherits instance variables, methods, and additional component flavors from the components. In a well-organized program, each component flavor defines a single facet of behavior. When two types of objects have some behavior in common, they each inherit it from the same flavor. This code need not be duplicated.

In summary, each real-world object is modelled by a single Lisp object. The object's flavor defines the inherent structure of the object. The state of each individual object is stored in its instance variables. Generic functions are used to perform operations on flavor instances. Each generic function is implemented with one or more methods; each method performs the operation on objects of a certain flavor.

3.2.2 Concept of Generic Functions

Like ordinary functions, generic functions take arguments, perform an operation, and perhaps return useful values. The first argument to a generic function is an object (an instance of a flavor). Unlike ordinary functions, generic functions behave a certain way for objects of one flavor, and behave in another way for objects of another flavor.

For example, in writing a text editor we might define two flavors: **character** and **paragraph**. It is important to be able to erase characters and paragraphs, so we define a generic function called **erase**. When we use **erase** on a **character** object, we want the character to disappear from view, and not to be saved anywhere. However, when we use **erase** on a **paragraph** object, we want the paragraph to disappear, and we also want to save the paragraph in a buffer somewhere. This feature aids users in restoring large bodies of text to their buffers.

Using Flavors terminology, we implement the generic function by writing two *methods*. Both methods are associated with the generic function **erase**. One methods is associated with the **character** flavor; the other is associated with the **paragraph** flavor. When the generic function **erase** is called on an object, the flavor of the object determines which method is used.

Generic functions differ from ordinary functions in that each generic function can have several methods associated with it, and Flavors chooses which one to use on any given call by the flavor of the first argument. An ordinary function has a single body of code that is always executed when the function is called.

For further discussion: See the section "Generic Functions", page 367. See the section "Using Message-Passing Instead of Generic Functions", page 471.

3.2.3 Concept of Message-passing

In previous versions of Flavors, the only mechanism for operating on objects was called *message-passing*. Using message-passing, you can operate on an object by sending it a message. The object receives the message and selects the appropriate method to execute. You use the function **send** to send the message and **defmethod** to write methods for messages. In most cases the name of the message is a keyword.

In Genera 7.0, generic functions were introduced as the new and preferred way to operate on objects. Generic functions are smoothly integrated into the Lisp environment. Ordinary functions and generic functions are called with the same syntax. Making generic functions syntactically and semantically compatible with ordinary functions has the following advantages:

- The caller of a function need not know whether it is generic.
- The Common Lisp package system can be used to isolate modules and to distinguish between public and private interfaces by **export**ing the names of public generic functions.
- Debugging tools such as **trace** can be used on generic functions.
- They are true Lisp functions that can be passed as arguments and used as the first argument to **funcall** and **mapcar**:

```
(mapc #'reset counters)
```

It is important to continue to support message-passing because a large body of customer code and Symbolics system code had been developed using message-passing. There is generally not much point to converting existing code from message-passing to generic functions. However, when writing new programs, it is good practice to use generic functions instead of message-passing.

For more information on message-passing: See the section "Using Message-Passing Instead of Generic Functions", page 471.

3.2.4 Simple Use of Flavors

This section illustrates the basic concepts of using flavors. For a lengthier example: See the section "Example of Programming with Flavors: Life", page 387.

Representing Objects

The program we are writing deals with ships. We must first determine a way to represent ships. If the important things to know about a ship are its name, x-velocity, y-velocity, and mass, we can represent ships as follows:

```
(defflavor ship (name x-velocity y-velocity mass)
           ()                         ; no component flavors
  :readable-instance-variables
  :writable-instance-variables
  :initable-instance-variables)
```

This **defflavor** form defines a flavor that represents ships. The name of the flavor is **ship**. The instance variables are **x-velocity**, **y-velocity**, and **mass**. The empty list could contain component flavors to be mixed into the definition of **ship**; in this case, **ship** has no component flavors. The form contains three options, which have the following effects:

:readable-instance-variables
Defines accessor functions that enable you to query the object for the value of instance variables. In this case four functions are automatically generated: **ship-name**, **ship-x-velocity**, **ship-y-velocity**, and **ship-mass**.

:writable-instance-variables
Enables you to alter the value of instance variables using **setf** and the accessor functions. When this option is supplied, the instance variables are also made **:readable-instance-variables**.

:initable-instance-variables
Enables you to initialize the value of an instance variable when you make a new instance.

The **ship** flavor is a framework, and many ships will fit into that framework. We represent each real-life ship as an instance of the **ship** flavor. Each instance stores information about one particular ship in its instance variables.

To create instances, we use **make-instance** as follows:

```
(setq my-ship (make-instance 'ship :name "Titanic"
                                   :mass 14
                                   :x-velocity 24
                                   :y-velocity 2))
```

As a result of giving the **:initable-instance-variables** option to **defflavor**, we were able to initialize the values of the instance variables when making the instance of **ship**. The symbol **my-ship** is now bound to the newly created instance.

Operating on Objects

We can query **my-ship** for the value of any of its instance variables by using a function that was automatically generated as a result of the **:readable-instance-variables** option to **defflavor**. For example:

```
(ship-name my-ship)
--> "Titanic"
```

Similarly, because we included the **:writable-instance-variables** option, we can change the value of an instance variable. For example:

```
(setf (ship-mass my-ship) 100)
--> 100
```

We can examine the instance by using **describe**:

```
(describe my-ship)
-->#<SHIP 54157652>, an object of flavor SHIP,
   has instance variable values:
   NAME                  "Titanic"
   X-VELOCITY:           24
   Y-VELOCITY:           2
   MASS:                 100
```

We can define new operations (called generic functions) for instances of the **ship** flavor, using **defmethod**. Inside the body of the method, we can access the instance variables of the object by name. For example:

```
(defmethod (speed ship) ()
  (sqrt (+ (expt x-velocity 2)
           (expt y-velocity 2))))
```

To the caller, a generic function is just like any other Lisp function:

```
(speed my-ship)
-->24.083189
```

Operating on Different Kinds of Objects with One Generic Function

Generic functions are more interesting when they can be used to operate on different kinds of objects. Let's introduce a new flavor, **comet**, and create an instance of it:

```
(defflavor comet (x-velocity y-velocity z-velocity)
          ()
  :initable-instance-variables)

(setq my-comet (make-instance 'comet
                              :x-velocity 312
                              :y-velocity 23.5
                              :z-velocity 26))
```

We can define a new method that implements the **speed** generic function on instances of **comet**:

```
(defmethod (speed comet) ()
  (sqrt (+ (expt x-velocity 2)
           (expt y-velocity 2)
           (expt z-velocity 2))))
```

To find the speed of my-comet:

```
(speed my-comet)
-->313.9622
```

The generic function **speed** now has two different methods defined for it. One method implements **speed** on **ship** objects, the other on **comet** objects. When you call the generic function **speed** on an object, Flavors determines the flavor of that object and chooses the appropriate method for it.

Mixing Flavors

For a simple example of mixing flavors, we can represent a passenger ship. A passenger ship has the same characteristics as the **ship** flavor, with one additional attribute: a list of passengers. We can use the **ship** flavor as a building block for the new flavor, as follows:

```
(defflavor passenger-ship (passenger-list)
           (ship)
  :initable-instance-variables)
```

The **ship** flavor is called a *component flavor* of **passenger-ship**. **passenger-ship** inherits instance variables and methods from **ship**. For example, when we make an instance of **passenger-ship**, we can initialize **name**, **mass**, **x-velocity**, and **y-velocity**, all instance variables inherited from **ship**:

```
(setq my-passenger-ship
        (make-instance 'passenger-ship
                                :name "QE2"
                                :mass 450
                                :x-velocity 12
                                :y-velocity 0
                                :passenger-list '(Brown Jones Lee)))
```

Similarly, we can use the generic function **speed** on **passenger-ship**; the method was inherited from the component flavor **ship**.

```
(speed my-passenger-ship)
-->12
```

3.2.5 Motivation for Using Flavors

The motivation for using flavors usually arises in large programs. Flavors enable you to organize programs around *objects*, which model real-world things. An object has a *state* and *operations* that can be performed on it. Flavors can be considered an extension of the Common Lisp facility for defining new structures with **defstruct**.

Here are some guidelines for using flavors:

- When you would consider using **defstruct**.
- When your program contains lots of particular kinds of objects.
- When different kinds of objects share some characteristics.
- When one operation is appropriate for different kinds of objects.
- When you want to define a protocol that different programs can use.

The last item illustrates the strengths of Flavors:

For example, we could implement output streams as flavors. The "protocol" consists of a set of functions that are guaranteed to work on any output stream. These functions might include **output-char**, **output-string**, and **output-line**, among others.

This protocol makes it easy to write programs that appear device-independent, by using the generic functions available for output streams. The use of the generic functions is the same, no matter how the actual output is implemented.

From the other perspective, you can implement a new kind of output device by implementing all the operations handled by output streams. Then all existing programs that deal with output streams work on the new device.

```
Various                                        Various
Programs              Single Protocol          Output Devices

                    +------------------+
User programs       |   output-char    |   Console
Hardcopy            |   output-string  |   Laser printer
Document Examiner   |   output-line    |   ...
                    +------------------+
```

Using Flavors frees programs from needing to understand how each output operation is implemented on the different devices. This style of programming is modular, easy to extend, and easy to maintain.

3.2.6 Comparing defstruct Structures and Flavors

This section compares and contrasts **defstruct** structures and flavors.

Flavors and **defstruct** enable you to:

- Use object-oriented programming in the Lisp environment, encouraging a modular style of programming.
- Create and use new aggregate data types.
  - ° Elements of the new data types are named.
- Specify that functions should be automatically generated to read and write the elements of the new data types.
- Include the definition of one new data type in another:
  - ° Elements are inherited from an existing structure.
  - ° Functions for reading and writing elements are inherited.

The major differences between flavors and **defstruct** structures are as follows:

defstruct *Structures*	*Flavors*
Each structure can have only one component structure (given with **:include**).	You can mix flavors liberally, and include many flavor components in the definition of a new flavor.
A structure does not inherit operations from its component structure.	A flavor inherits methods for operations from its component flavors.
It is difficult, inconvenient, and sometimes impossible to add, delete, or rename slots.	It is easy and convenient to add, delete, and rename instance variables, or to change other flavor characteristics.
You can control the internal representation of the structure, such as a list, array, or other representation.	You cannot control the internal representation of a flavor.
It is somewhat faster to	It is somewhat slower to

reference a slot of a **defstruct** structure.	access instance variables than slots.
You can cache a **defstruct** structure in an array register.	You cannot cache an instance in an array register.
	Flavors offers many advanced features and programming practices that are not available using **defstruct**.

3.3 Overview of the Table Management Facility

A *table* is a data structure that consists of some number of *entries*, each containing one or more objects. The number of objects per entry is fixed and uniform in any given table. The simplest tables consist of entries which are *keys*. In the most common table, the first object in each entry of a table is the key, and the second object is the *value*. In more complex tables, there can be some combination of multiple keys and multiple values.

This sample table is made up of key and value pairs, where the key is the bird type and the value is a list of foods that bird eats:

	KEY (bird)	VALUE (diet)
	blue-heron	(frogs snakes turtles)
ENTRY	horned-owl	(mice snakes)
	pelican	(fish)
	...	...

The principal operations on tables are:

- searching by key
- inserting and deleting entries
- examining all entries
- deleting all entries

Some tables also support the additional operations of retrieving the first entry, retrieving the last entry, and possibly retrieving the entries in order by key.

The table management facility performs these operations on tables of many forms, using one common interface. Thus, you need not worry about the internal representation of the data or other properties of the table. If you create tables with this facility, you have code which is easily ported to Common Lisp while taking advantage of the efficiencies provided by the facility. If you create tables which do not use the Symbolics extensions to the **make-hash-table** function, your code will already be compatible with Common Lisp.

You create table objects with the **make-hash-table** function, and add new entries by using a combination of the **gethash** function and the **setf** macro. Here is a simple example:

```
(setq bird-table (make-hash-table :size 10))
   => #<EQL-ALIST-PROCESS-LOCKING-DUMMY-GC-LOCKING-
        ASSOCIATION-MUTATING-TABLE 35646610>

(setf (gethash 'wader bird-table) 'flamingo) => FLAMINGO

(setf (gethash 'raptor bird-table) 'bald-eagle) => BALD-EAGLE

(hash-table-count bird-table) => 2

(describe bird-table)
   => #<EQL-ALIST-PROCESS-LOCKING-DUMMY-GC-LOCKING-
        ASSOCIATION-MUTATING-TABLE 340050026> is a table with 2
        entries.
   Do you want to see the contents of the hash table? (Y or N) Yes.
   Do you want it sorted? (Y or N) Yes.
   Test function for comparing keys = EQL, hash function =
        CLI::XEQLHASH
     RAPTOR → BALD-EAGLE
     WADER → FLAMINGO
   #<EQL-ALIST-PROCESS-LOCKING-DUMMY-GC-LOCKING-ASSOCIATION-
     MUTATING-TABLE 340050026>
```

In this example, the keys are **wader** and **raptor**, and the associated values are **flamingo** and **bald-eagle**. Each entry in the table associates a bird type to a bird name.

The table management facility is based on Flavors, and defines a large family of table flavors and generic functions for accessing them. This allows for easy use, as well as flexibility and extensibility.

4. Programs and Their Management: Evaluation, Scoping, Flow of Control, Packages, Conditions

4.1 Overview of Evaluation

Evaluation is the process of recursively executing Lisp forms and returning their values. Simply put, evaluation is the computation performed by a program. A form is passed to the evaluator. If the form is a *symbol*, the evaluator returns the *binding* (*value*) of the symbol. If the symbol has no binding, the evaluator signals an error. If the form is a list, the evaluator looks first at the **car** of the list. If the **car** is a symbol, it retrieves the functional value of that symbol. If that functional value is a function definition, the remaining forms in the list are evaluated in turn and then the function is applied to the result to produce the final value of the list. If the **car** of the list is another list, the **car** of that list is evaluated, and so on.

For example, if the evaluator is given **(+ 4 5)**, it determines first that the form is a list. Then it looks at the +. It retrieves the functional value of this symbol, which is the addition function. It next looks at the **4**, which has as its value 4; then it looks at **5**, which is 5; and finally it applies the addition function to 4 and 5 which produces 9. It then returns **9** as the value of **(+ 4 5)**. This is indicated in the documentation like this:

```
(+ 4 5) => 9
```

4.2 Overview of Dynamic and Lexical Scoping

Scoping refers to the range of the environment in which a variable exists and can be used in computation. There are two kinds of scoping, *dynamic scoping* and *lexical scoping*. If a variable has dynamic scope (that is, has been declared *special*) it can be used in computation anywhere for as long as it exists, that is, from the time it is bound until it is explicitly unbound. (See the section "Special Forms for Defining Special Variables", page 502.) If a variable has lexical scope, it can only be used in computation within the textual confines of the Lisp form that defines it.

For example:

```
(defun mapc (funct list)
  (loop for x in list do    ;x is bound here
    (funcall funct x)))
```

```
(defun print-long-strings (strings x)    ;x is bound here
  (mapc #'(lambda (str)
            (if (> (length str) x)    ;which x is this?
              (print str)))
        strings))
```

In the definition of **mapc**, **x** is defined. Another **x** is defined as one of the arguments to **print-long-strings**. In the computation performed by the **lambda** there is a reference to **x**.

If **x** has dynamic scope, the reference to **x** in the function **print-long-strings** refers to the **x** in **mapc** because the loop in **mapc** is executing when the reference to **x** is made and the **x** in that loop is thus the most recently bound **x**. (This is probably not what the programmer intended.)

If **x** is lexically scoped, the **x** in **mapc** only exists as **x** inside the textual definition of **mapc**. Inside the textual definition of **print-long-strings**, the **x** refers to the argument to **print-long-strings**.

User defined variables in Symbolics Common Lisp are lexically scoped unless you explicitly declare them special.

4.3 Overview of Flow of Control

Symbolics Common Lisp provides a variety of structures for controlling program flow. A *conditional* construct is one that allows a program to make a decision, and do one thing or another based on some logical condition. Local and nonlocal *exits* allow the transfer of control from one section of a program to another. *Iteration* permits a programmer to execute a command multiple times.

The simplest conditional form is the **if**-*then* form, which can be extended to the **if**-*then-else* form. An example of this two-way conditional would be as follows :

```
(if (= 1 2) "equal" "not equal") => "not equal"
```

The logical forms **and**, **or**, and **not** let you build multi-way conditional constructs. A multi-way conditional is often equivalent to an **if**-*then-else-else...* form, but it can be clearer, more compact, and easier to read than a long line of *else* statements.

The most basic multi-way conditional is **cond**, consisting of the symbol **cond** followed by several *clauses*. Each clause represents a case that is selected if its antecedent is satisfied and the antecedents of all preceding clauses were not satisfied.

For example:

```
(cond ((and (equal "day" "day") (= 1 2)) "star light")
      ((> 1 2) "prefix or postfix")
      (t "drop out")) => "drop out"
```

Note the use of **t** in the last clause as a "use if all else fails" provision.

Premature exit from a piece of code is another mechanism for controlling program flow. Depending on their *scope* (the spatial or textual region or the program within which references can occur), exits can be *local* or *nonlocal.*

block and **return-from** are the primitive special forms for *local exit* from a piece of code. **block** defines a program portion that can be safely exited at any point, and **return-from** does an immediate transfer of control to exit from **block**. Local exits have *lexical* scope, that is, **block** and **return-from** can only operate within the portion of code textually contained in the construct that establishes them.

catch and **throw** are the special forms used for *nonlocal exits.* **catch** evaluates forms; if a **throw** is executed during the evaluation, the evaluation is immediately aborted at that point and **catch** immediately returns a value specified by **throw**. Nonlocal exits have *dynamic* scope, that is, the catch/throw mechanism works even if the **throw** form is not textually within the body of the **catch** form.

The repetition of an action (usually with some changes between repetitions) is called *iteration,* and is provided as a basic control structure in most languages. *Recursion* is one alternative to iteration. This programming method has the function call itself, thus causing an iteration. Recursion is analogous to mathematical induction.

Here is a very simple example of recursion:

```
;; Two things are defined as EQUALX if they are either EQ, or if
;; they are lists containing EQUALX elements.
;; Therefore EQUALX calls EQUALX recursively.

(defun equalx (a b)
  (cond ((eq a b) t)
        (t
         (and (listp a)
              (listp b)
              (equalx (car a) (car b))
              (equalx (cdr a) (cdr b))))))
```

This example uses recursion to traverse a tree:

```
(defun max-fringe (tree)
  (if (atom tree)
      tree
    (max (max-fringe (car tree))
         (max-fringe (cdr tree)))))
```

Recursion has some important advantages and disadvantages with respect to iteration.

Symbolics Common Lisp provides three styles of iteration: *mapping*, **do** and **loop**.

Mapping is a type of iteration in which a function is successively applied to pieces of a list. The result of the iteration is a list containing the respective results of the function application.

Mapping is used when a problem is easily expressed by a function followed by any number of lists.

For example:

```
(map 'list #'+ '(1 2 3 4) '(2 3 1 4)) => (3 5 4 8)
```

The use of mapping results in clear and concise code.

For more general iteration than mapping, you can use the simplest form of iteration, the **do** form. **do** provides a generalized iteration facility, with an arbitrary number of "index variables" whose values are saved when the **do** is entered and restored when it is left, that is, they are bound by the **do**. **do** is simple to use; however, it is often quite hard to read later.

For example:

```
(do  ((i 0 (+ 1 i))                         ; searches list for Dan.
      (names '(Fiona Tiffany Jen Kristen Wendy Sandy Dan Tom)
             (cdr names)))
     ((null names))
  (if (equal 'Dan (car names))
      (princ "Hi Jen"))) => Hi Jen
NIL
```

Even more simple and flexible than **do** is the **loop** macro which provides a programmable iteration facility. The basic structure of a loop is as follows:

(**loop** *iteration clauses*
          do
     body)   `; loop alone returns` **nil**

The *iteration clauses* control the number of times the *body* will be executed. When any iteration clause finishes, the body stops being executed. The word **do** is the keyword which introduces the body of loop.

The general approach is that a form introduced by the word **loop** generates a single program loop, into which a large variety of features can be incorporated. These features work by means of keywords, of which there is a large number. Note that **loop** keywords are not prefixed with a colon (:) character. Keywords like **repeat** or `(for` *x* `from` ...) for instance let you control the number of times through an iteration. Other keywords such as `(collect` *x* `into` *num*) let you accumulate a return value for the iteration. Most of the keywords for **loop** are Symbolics Common Lisp extensions to the language specification in Guy L. Steele's *Common Lisp: the Language.*

The use of some loop keywords might be as follows :

```
(loop repeat 5
      do
  (princ "hi ")) => hi hi hi hi hi
NIL

(loop for x from 1 to 5 by 1
      with y = 9
      initially (princ y)
      do
  (princ x))  => 912345
NIL

(loop for x in '(a d c e)
      do
  (princ x)) => ADCE
NIL
```

The order of **loop** keywords is mostly a matter of taste and style. Many of them are accepted in several synonymous forms (for example, **collect** and **collecting**), to let you write code that looks like stylized English. Use of the appropriate keywords helps to write code that is easier to read.

Not so clear:

```
(loop as x to 4 by 1 from 1
      collect x into num
      finally (return num))  => (1 2 3 4)
```

Better:

```
(loop for x from 1 to 4 by 1
      collect x into num
      finally (return num)) => (1 2 3 4)
```

In more advanced uses of iteration it is possible to define your own iteration paths, that is, to build your own iteration-driving clauses.

4.4 Overview of Packages

Lisp programs are made up of function definitions. Each function has a name to identify it. Names are symbols. (See the section "Overview of Symbols", page 14.) Each symbol can have only one function definition associated with it, so names of functions must be unique or else the behavior of a program would be completely unpredictable.

For example, if the compiler has a function named **pull**, and you load a program that has its own function named **pull**, the function definition of the symbol **pull** gets redefined to be that of the program just loaded, probably breaking the compiler. (Of course, Genera displays a warning message when such a redefinition happens.)

Now, if two programs are to coexist in the Lisp world, each with its own function **pull**, then each program must have its own symbol named "**pull**". The same reasoning applies to any other use of symbols to name things. Not only functions but variables, flavors, and many other things are named with symbols, and hence require that a program have its own collection of these symbols.

Since programs are written by many different people who do not get together to insure that the names they choose for functions are all unique, programs are isolated from each other by *packages.*

A *package* is a mapping from names to symbols. Two programs can use separate packages to enable each program to have a different mapping from names to symbols. In the example above, the compiler can use a package that maps the name **pull** onto a symbol whose function definition is the compiler's **pull** function. Your program can use a different package that maps the name **pull** onto a different symbol whose function definition is your function. When your program is loaded, the compiler's **pull** function is not redefined, because it is attached to a symbol that is not affected by your program. The compiler does not break.

For example, continuing with **pull**, the compiler has its symbols in the **compiler** package, so its **pull** function would be **compiler:pull.** If you have defined a package **mypackage** for your program, your **pull** function is **mypackage:pull.** Functions within each package can just refer to **pull** and get the right function, since the other **pull** would need its package prefix.

Two programs that are closely related might need to share some common functions. For example, a robot control program might have a function called **arm** that moves the robot arm to a specified location. A second program, a blocks world program, might want to call **arm** as part of its **clear** function that removes blocks from the top of a block to be picked up. If the robot control program is in the **robot** package, and the blocks world program is in the **blocks** package, the blocks world program can refer to the arm function by calling it as **robot:arm.** However, the blocks world is likely to need **arm** frequently, and calling it as

robot:arm is tedious for a programmer. The blocks world program really needs to have the function **arm** in its own package. In fact, the **robot** package probably contains many functions the blocks world program needs, so the blocks world program wants to have the **robot** package available in its own **blocks** package.

The package a symbol is defined in is called its *home package.* The symbols in a package can be designated as *internal* (belonging only to that package) or *external* (available to other packages, as in the **robot:arm** example). External symbols are said to be *exported.* Symbols that are exported can be *imported* by another package. If a program needs to share most or all of the external symbols in another package, it can import all the external symbols of that package. This is called *using* the package.

Sharing does have some disadvantages, however. To continue with the **robot:arm** example, if the blocks world program were to decide to define its own **arm** function while it was using the **robot** package, this would redefine **arm** in the **robot** package as well. This is because sharing symbols means that now the **robot** package and the **blocks** have the same pool of symbols. For more details on sharing and its consequences: See the section "Qualified Package Names", page 648.

Genera sets up a package for you called **cl-user.** This is the default package of your Lisp Listener. **cl-user** *uses* **common-lisp-global** so all the functions of Common Lisp are available to your program. When you define your own package for your program, you can designate, using the **use-package** function or the **import** function, those symbols from other packages that your program needs. For information about packages defined in Genera: See the section "System Packages", page 663. You can also declare which symbols in your package are external (can be imported or used by other packages) and which are internal (for your program alone). For information about defining your own package: See the function **make-package** in *Symbolics Common Lisp: Language Dictionary.*

Since using another package might possibly result in a name conflict (the package you are using might have a symbol of the same name as one in your package), the system checks and warns you of any conflicts. You can select which symbol your program uses. This process is called *shadowing.* The **shadow** or **shadowing-import** functions control whether the symbol in your package or the imported symbol is the one to be used. Shadowing is a complex process. For more information about it: See the section "Shadowing Symbols", page 642.

4.5 Overview of Conditions

Conditions is an advanced topic geared to programmers who want to customize the error handling mechanism.

The documentation describes the following major topics:

- Mechanisms for handling conditions that have been signalled by system or application code.
- Mechanisms for defining new conditions.
- Mechanisms that are appropriate for application programs to use to signal conditions.
- All of the conditions that are defined by and used in the system software.

Symbolics Common Lisp condition handling is based on flavors, which are an extension of the Common Lisp language. Here are some basic topics and the terminology associated with them.

Event	An event is "something that happens" during the execution of a program. It is some circumstance that the system can detect, such as the effect of dividing by zero. Some events are errors – which means something happened that was not part of the contract of a given function – and some are not. In either case, a program can report that the event has occurred, and it can find and execute user-supplied code as a result.
Condition	Each standard class of events has a corresponding flavor called a condition. For example, occurrences of the event "dividing by zero" correspond to the condition **sys:divide-by-zero**. Sets of conditions are defined by the flavor inheritance mechanism. The symbol **condition** refers to all conditions, including simple, error, and debugger conditions.
Simple conditions	These are built on the basic flavor **condition**.
error conditions	A base flavor for many conditions. Refers to the set of all *error* conditions.
Debugger conditions	Conditions built on the flavor **dbg:debugger-condition**. They are used for entering the Debugger without necessarily classifying the event as an error. This is intended primarily for system use.
Signalling	The mechanism for reporting the occurrence of an event. The signalling mechanism creates a *condition object* of the flavor appropriate for the event. The condition object is an instance of that flavor, which contains information about the event, such as a textual message to report, and various parameters of the condition. For example, when

	a program divides a number by zero, the signalling mechanism creates an instance of the flavor **sys:divide-by-zero**. You can signal a condition by calling either **signal** or **error**.
Handling	The processing that occurs after an event is signalled.
Handler	A piece of user-supplied code that is bound with a program for a particular condition or set of conditions. When an event occurs, the signalling mechanism searches all of the currently bound handlers to find the one that corresponds to the condition. The handler can then access the instance variables of the condition object to learn more about the condition and hence about the event. Genera includes default mechanisms to handle a standard set of events automatically.
Proceeding	After a handler runs, the program might be able to continue execution past the point at which the condition was signalled, possibly after correcting the error.
Restart	Any program can designate *restart points*. After a handler runs, the restart facility allows a user to retry an operation from some earlier point in the program.

PART II.

Data Types

5. Data Types and Type Specifiers

Symbolics Common Lisp provides a variety of data object types, as well as facilities for extending the type hierarchy. It is important to note that in Lisp it is data objects that are typed, not variables: any variable can have any Lisp object as its value.

5.1 Hierarchy of Data Types

In Symbolics Common Lisp, a data type is a (possibly infinite) set of Lisp objects. The data types defined in Symbolics Common Lisp are arranged into a hierarchy (actually a partial order) defined by the subset relationship.

A type called **common** encompasses all the data objects required by the Common Lisp language. The set of all objects in Symbolics Common Lisp is specified by the symbol **t**. The empty data type, which contains no objects, is denoted by **nil**.

The following terminology expresses the defined relationships between data types.

If x is a *supertype* of y, then any object of type y is also of type x, and y is said to be a *subtype* of x. For example, the type **integer** is a subtype of **rational**. The type **t** is a supertype of every type whatsoever: every object belongs to type **t**. The type **nil** is a subtype of every type whatsoever: no object belongs to type **nil**.

If type x and y are *disjoint*, then no object can be both of type x and of type y. For instance, the types **integer** and **ratio** are disjoint subtypes of **rational**.

Types a_1 through a_n are an *exhaustive union* of type x if each a_j is a subtype of x, and any object of type x is necessarily of at least one of the types a_j; a_1 through a_n are furthermore an *exhaustive partition* if they are also pairwise disjoint. The types **cons** and **null** form an exhaustive partition of the type **list**.

Figure 1 shows the data type hierarchy for Symbolics Common Lisp as a tree whose root is the type **t**. Data types linked by connecting lines are related in a supertype-subtype relationship. Data types with no explicit connecting lines are not necessarily disjoint.

Certain objects such as the set of numbers or the set of strings are identified by associated symbolic names or lists, called *type specifiers*. See the section "Type Specifiers", page 73.

Since many Lisp objects belong to more than one such set, it doesn't always make sense to ask what the *type* of an object is; instead, one usually asks only whether an object *belongs* to a given type. The predicate **typep** tests a Lisp object against one of the standard type specifiers to determine if it belongs to that type.

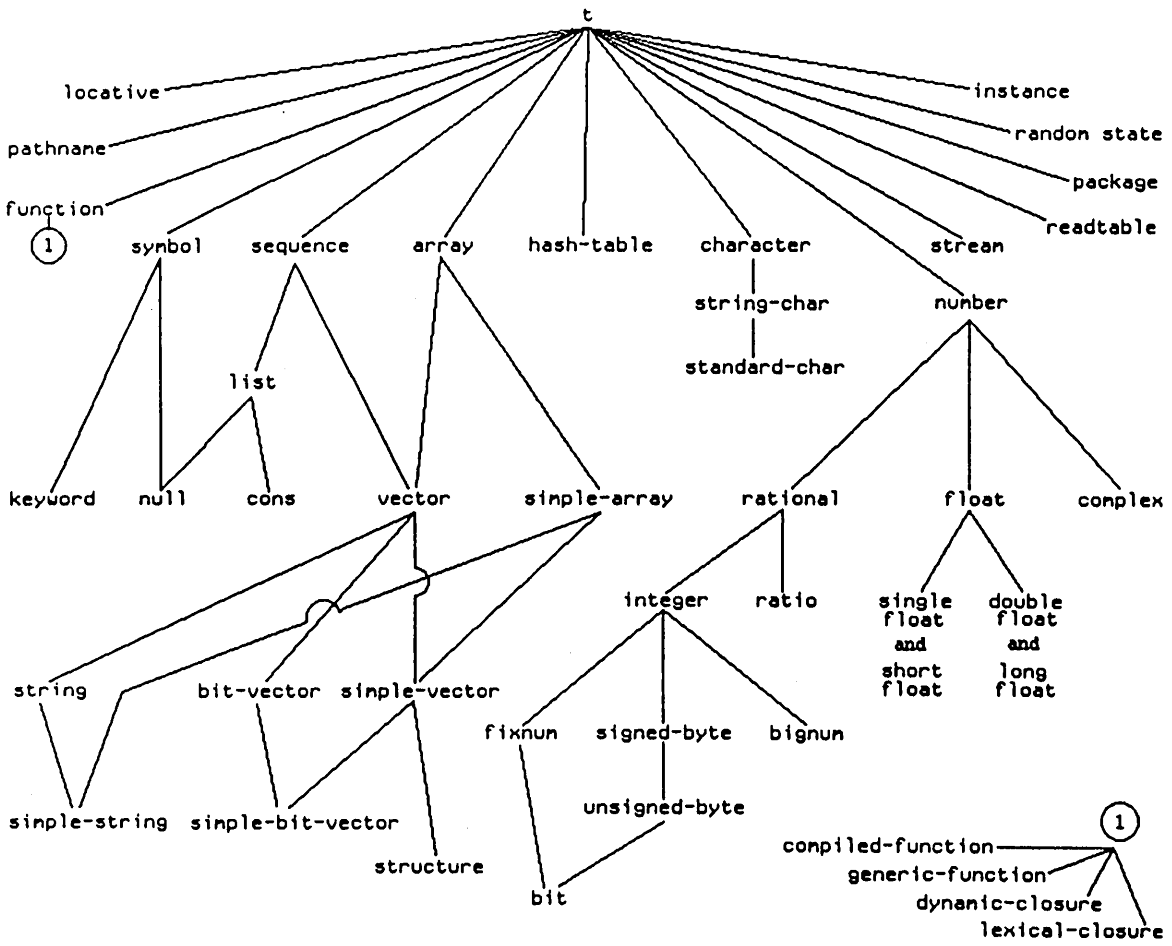


Figure 1. Symbolics Common Lisp Data Types

5.2 Some Major Data Types

Here are brief descriptions of the top level and a few lower-level Symbolics Common Lisp data types. The remainder of this manual covers the complete set of data types and their operations in detail.

- *Numbers* are provided in several forms and representations. Symbolics Common Lisp provides a true integer data type: any integer, positive, or negative, has in principle a representation as a Symbolics Common Lisp data object, subject only to total memory limitations, rather than to machine word width. A true rational data type is provided: the quotient of two integers, if not an integer, is a ratio. Floating-point numbers of single and double precision are also provided, as well as Cartesian complex numbers.

- *Characters* represent printed glyphs such as letters or text formatting operations. Strings are one dimensional arrays of characters. Symbolics Common Lisp provides for a rich character set, including ways to represent characters of various type styles.

- *Symbols* are named data objects. Lisp provides machinery for locating a symbol object, given its name (in form of a string). Symbols have property lists, which in effect allow symbols to be treated as record structures with an extensible set of named components, each of which may be any Lisp object. Symbols also serve to name functions and variables within programs.

- *Cons* is a primitive Lisp data type that consists of a *car* and a *cdr*. Linked conses are used to represent a non-empty list.

- *Sequences* are instances of the sequence type. A sequence is a supertype of the *list* and *vector* (one-dimensional array) types. These types have the common property that they are ordered sets of elements. Sequence functions can be used on either lists or vectors.

- *Lists* are represented in the form of linked cells called conses. The *car* of the list is its first element; the *cdr* is the remainder of the list. There is a special object (the symbol **nil**) that is the empty list. Lists are built up by recursive application of their definition.

- *Arrays* are dimensioned collections of objects. An array can have a non-negative number of dimensions, up to eight, and is indexed by a sequence of integers. A general array can have any Lisp object as a component; other types of arrays are specialized for efficiency and can hold only certain types of Lisp objects. It is possible for two arrays, possibly with differing dimension information, to share the same set of elements (such that modifying one array modifies the other also) by causing one to be displaced to the other. One-dimensional arrays of any kind are called vectors. One-dimensional arrays of characters are called strings. One-dimensional array of bits (that is, of integers whose values are 0 or 1) are called bit-vectors.

- *Tables* provide an efficient way of associating Lisp objects. This is done by associating a key with a value. Some tables are *hashed*, which is a method for storing the association between the key and the value; this permits faster association in exchange for some storage overhead.

- *Readtables* are data structures used to control the parsing of expressions. This structure maps characters into syntax types. This is extensively used by macro characters to read their definitions. You can reprogram the parser to a limited extent by modifying the readtable.

- *Packages* are collections of symbols that serve as name spaces. The parser recognizes symbols by looking up character sequences in the current package.

- *Pathnames* represent names of files in a fairly implementation-independent manner. They are used to interface to the external file system. For a discussion of pathnames: See the document *Reference Guide to Streams, Files, and I/O.*

- *Streams* represent sources or sinks of data, typically characters or bytes. They are used to perform I/O, as well as for internal purposes such as parsing strings. For a discussion of streams: See the document *Reference Guide to Streams, Files, and I/O.*

- *Random-states* are data structures used to encapsulate the state of the built-in random-number generator.

- *Flavors* are user-defined data structures. **defflavor** is used to define new flavors. The name of the flavors becomes a valid type symbol; it is a subtype of **instance**. When flavors are built from components, the more specific flavors are subtypes of their component flavors.

- *Structures* are user-defined record structures, objects that have named components. The **defstruct** facility is used to define new structure types. The name of the new structure type becomes a valid type symbol.

- *Functions* are objects that can be invoked as procedures; these may take arguments and return values. (All Lisp procedures return values and therefore every procedure is a function.) Such objects include compiled-functions (compiled code objects). Some functions are represented as a list whose car is a particular symbol such as **lambda**. Symbols can also be used as functions.

- *Compiled-functions* The usual form of compiled, executable code is a Lisp object called a "compiled function." A compiled function contains the code for one function. Compiled functions are produced by the Lisp Compiler and are usually found as the definitions of symbols. The printed representation of a compiled function includes its name, so that it can be identified. About the only useful thing to do with compiled functions is to *apply* them to arguments. However, some functions are provided for examining such objects, for user convenience.

- *Generic functions* are functions that operate on flavor instances. They can be defined explicitly with **defgeneric**, or implicitly with **defmethod**.

- *Lexical Closure* is a functional object that contains a *lexical* evaluation environment, for example, an internal lambda in an environment containing lexical variables. These variables can be accessed by the environment of the internal lambda; the closure is said to be a closure of the free lexical variables. Invocation of a lexical closure provides the necessary data linkage for a function to run in the environment in which the closure was made.

- *Dynamic Closure* is a functional object that contains a *dynamic* evaluation environment. Dynamic Closures are created by the **closure** function and the **let-closed** special form. Dynamic Closures are closures over *special* variables. Invocation of a dynamic closure causes special variables to be bound around the closed-over function.

- *Locative* is a Lisp object used as a pointer to a single memory cell in the system. Locatives are a low-level construct, and as such, most programmers never use them.

These categories are not always mutually exclusive.

5.3 Type Specifiers

A *type specifier* is a symbol or a list naming Lisp objects. Symbols represent predefined classes of objects, whereas lists usually indicate combinations or specializations of simpler types. Symbols or lists can also be abbreviations for types that could be specified in other ways. The various type-checking functions can be applied to type specifiers regardless of whether they are symbols or lists. See the section "Determining the Type of an Object", page 79.

Note that although type specifiers and functions sometimes share the same name, they work differently and should not be confused with each other.

5.3.1 Type Specifier Symbols

The predefined Symbolics Common Lisp type symbols include those shown in the table below. In addition, when a structure type is defined using **defstruct**, or a flavor is defined using **defflavor**, the name of the structure type and the flavor name respectively become valid type symbols. For more on individual symbols: See the section "Dictionary of Data Type Specifiers and Functions".

array	instance	short-float
atom	integer	simple-array
bignum	keyword	simple-bit-vector
bit	list	simple-string
bit-vector	lexical-closure	simple-vector
character	locative	single-float
common	long-float	signed-byte
compiled-function	nil	standard-char
complex	null	stream
cons	number	string
double-float	package	string-char
dynamic-closure	pathname	structure
fixnum	random-state	symbol
float	ratio	t
function	rational	unsigned-byte
generic-function	read-table	vector
hash-table	sequence	

5.3.2 Type Specifier Lists

Type specifier lists allow further combinations or specializations of existing data types, for example: denoting a list of objects that satisfy a type-checking predicate, declaring and/or defining specialized forms of data types, or constructing abbreviated forms of type specifiers.

The valid type specifier list formats are listed in the tables that follow the syntax section. For more on individual items: See the document *Symbolics Common Lisp: Language Dictionary*.

5.3.2.1 Type Specifier List Syntax

If a type specifier is a list, the first element of the list is a symbol, and the rest of the list is subsidiary type information. The symbol can be one of the standard type specifier symbols previously listed, but other symbols can also be used: symbols like **mod**, or **member** for example, work as type specifiers when used in type specifier lists, even though the symbols themselves are not type specifiers.

In many cases a subsidiary item can be *unspecified.* The unspecified subsidiary item is indicated by writing the symbol *. For example, to completely specify a vector type, one must mention the type of the elements and the length of the vector, as in:

```
(vector double-float 100)
```

To leave the length unspecified, one would write:

```
(vector double-float *)
```

To leave the element type unspecified, one would write:

```
(vector * 100)
```

Suppose that two type specifiers are the same, except that the first has an asterisk (*) where the second has a more explicit specification. Then the second denotes a subtype of the type denoted by the first.

As a convenience, if a list has one or more unspecified items at the end, such items can simply be dropped, rather than writing an explicit * for each one. If dropping all occurrences of * results in a singleton list, the parentheses can be dropped as well (the list can be replaced by the symbol in its **car**). For example, `(vector double-float *)` can be abbreviated to `(vector double-float)`, and `(vector * *)` can be abbreviated to `(vector)` and then simply to `vector`.

5.3.2.2 Predicating Type Specifiers

A type specifier list of the form

(satisfies *predicate-name*)

lets you define the set of all objects that satisfy the predicate named by *predicate-name*. *predicate-name* can be a symbol whose global function definition is a one-argument predicate, or a lambda-expression. (Note: allowing a lambda-expression for *predicate-name* is a Symbolics Common Lisp extension to Common Lisp.)

For example, the type

```
(satisfies numberp)
```

is the same as the type **number**. The call `(typep x '(satisfies p))` results in applying *p* to *x* and returning **t** if the result is true and **nil** if the result is false.

As an example, the type **string-char** could be defined as follows:

```
(deftype string-char ()
  '(and character (satisfies string-char-p)))
```

5.3.2.3 Type Specifier Lists That Combine

It is possible to define a data type in terms of other data types or objects. The following functions make up the appropriate type specifier list for this purpose:

(**member** &rest *list*)	An object is of this type if it is **eql** to one of the specified objects in *list*.
(**not** *type*)	Denotes the set of objects that are *not* of the specified *type*.
(**and** &rest *types*)	Denotes the intersection of the specified types.
(**or** &rest *types*)	Denotes the union of the specified types.

5.3.2.4 Type Specifier Lists That Specialize

You can construct type specifier lists that let you *declare* specialized forms of data types named by symbols. Such declarations allow optimization by the system. If the system actually creates that specialized form, the type specifier declaration results in further *discrimination* among existing data types.

Here is an example where the type specifier list serves for both *declaration* and *discrimination*. The list-format:

```
(array single-float)
```

permits the creation of a type of array whose elements are of type **single-float**. In other words, it declares to the array-creating function, **make-array** that elements will always be of the type **single-float**. Since Symbolics Common Lisp does create such specialized arrays, a test (using the predicate **typep**) of whether the array is actually of type `(array single-float)` returns **t**.

The valid list-format names for data types are listed below. Unless annotated to the contrary, each of the list-format names denotes specialized data types that can be created by Symbolics Common Lisp.

(**array** *element-type dimensions*)	Denotes the set of specialized arrays whose elements are all members of the type *element-type*, and whose dimensions match *dimensions*, a list of integers.
(**sequence** *type*)	Denotes the set of sequences whose elements are

all members of the type *type*. This is a Symbolics Common Lisp extension to Common Lisp.

(**simple-array** *element-type dimensions*)
Similar to (**array**...), additionally specifying that objects of the type are *simple* arrays.

(**vector** *element-type size*)
Denotes the set of specialized one-dimensional arrays whose elements are all of type *element-type*, and whose lengths match *size*.

(**simple-vector** *size*)
The same as (**vector** ...), additionally specifying that objects of the type are *simple* vectors. Declarative use only.

(**complex** *type*)
Every element of this type is a complex number whose real part and imaginary part are each of type *type*.

(**function** (*arg1-type arg2-type* ...) *value-type*)
Use this syntax for declaration. Every element of this type is a function that accepts arguments at *least* of the types specified by the *argj-type* forms, and returns a value that is a member of the types specified by the *value-type* form.

(**values** *value1-type value2-type* ...)
Use only as the *value-type* in a **function** type specifier or in a **the** special form. Specifies individual types when multiple values are involved.

5.3.2.5 Type Specifier Lists That Abbreviate

You can use type specifier list format to construct type specifiers that are abbreviations for other type specifiers. This is useful when the resulting type specifiers would be far too verbose to write out explicitly.

For those formats that specify a range such as *low* and *high*, each of these limits can be represented as an integer, a list of integers, or as the symbol *, meaning unspecified. The exact interpretation of the lower and upper limits depends on their representation: an integer is an *inclusive* limit; a list of an integer is an *exclusive* limit; the symbol * means that a limit does not exist and so effectively denotes minus or plus infinity, respectively.

Here are the valid formats:

(**number** *low-limit high-limit*)
Denotes the numbers between *low-limit* and *high-limit*. This is a Symbolics Common Lisp extension to Common Lisp.

(integer *low high***)**	Denotes the integers between *high* and *low*.
(mod *n***)**	Denotes the set of non-negative integers less than *n*.
(ratio *low high***)**	Denotes the set of ratios between *low* and *high*. This is a Symbolics Common Lisp extension to Common Lisp.
(signed-byte *s***)**	Denotes the set of integers that can be represented in two's-complement form in a byte of *s* bits. This is equivalent to `(integer -`2^{s-1} 2^{s-1} `- 1)`. `(signed-byte *)` is the same as **integer**.
(unsigned-byte *s***)**	Denotes the set of non-negative integers that can be represented in a byte of *s* bits. This is equivalent to `(integer 0 `2^{s}`-1)`. `(unsigned-byte *)` is the same as `(integer 0 *)`, the set of non-negative integers.
(rational *low high***)**	Denotes the rational numbers between *low* and *high* exclusive.
(float *low high***)**	Denotes the set of floating-point numbers between *low* and *high* exclusive.
(string *size***)**	Denotes the set of strings of the indicated *size*.
(simple-string *size***)**	Denotes the set of simple strings of the indicated *size*.
(bit-vector *size***)**	Denotes the set of bit-vectors of the indicated *size*.
(simple-bit-vector *size***)**	Denotes the set of simple-bit-vectors of the indicated *size*.

5.3.3 Defining New Type Specifiers

New type specifiers can come into existence in three ways. First, defining a new structure type with **defstruct** automatically causes the name of the structure to be a new type specifier symbol. Second, defining a new flavor type with **defflavor** automatically causes the name of the flavor to be a new type specifier symbol. Third, the **deftype** special form can be used to define new type-specifier abbreviations.

5.4 Type Conversion Function

The function **coerce** can be used to convert an object to an equivalent object of another type.

It is not generally possible to convert any object to be of any type whatsoever; only certain conversions are permitted, as summarized below. The dictionary entry for this function illustrates its operation more fully.

- Any sequence type can be converted to any other sequence type, provided the new sequence can contain all actual elements of the old sequence.
- Some strings, symbols, and integers, can be converted to characters.
- Any noncomplex number can be converted to a single- or double-floating-point number.
- Any number can be converted to a complex number.
- Any object can be coerced to type **t**.

5.5 Determining the Type of an Object

These general type-checking functions make it possible to test relationships between objects in the type hierarchy, determine if an object belongs to a given data type, get the type specifier list for standard data types, and identify equivalent data type descriptions.

Type-checking functions are useful, among others, in controlling program flow, and in error-checking.

There are also numerous specialized predicates for type-checking. See the section "Predicates", page 271. That section contains summary tables for all type-checking predicates. The individual chapters for each data type further discuss these predicates.

type-of *object*	Returns the most specific type specifier describing a type of which *object* is a member.
sys:type-arglist *type*	Returns a lambda-list of specifiers for *type*, if *type* is a defined Common Lisp type; returns a second boolean value, **t**, if *type* is a defined Common Lisp type, **nil** otherwise.
commonp *object*	Returns **t** if *object* is a Common Lisp data type.

typep *object type*	Returns **t** if *object* is of type *type*.
subtypep *type1 type2*	Returns **t** if *type1* is a subtype of *type2*.
equal-typep *type1 type2*	Returns **t** if *type1* and *type2* are equivalent type specifiers denoting the same data type.
typecase *object* &body *body*	Selects a clause for evaluation by determining if the type of an object matches a given data type. See the section "Conditionals", page 528.
ctypecase *object* &body *body*	"Continuable exhaustive type case." Like **typecase**, but signals a proceedable error if no clause is satisfied. See the section "Conditionals", page 528.
etypecase *object* &body *body*	"Exhaustive case." Like **typecase**, but signals a non-proceedable error if no clause is satisfied. See the section "Conditionals", page 528.
check-type *place type* &optional *type-string*	Signals an error if the contents of *place* are not of the specified *type*. See the section "Conditions", page 565.

5.5.1 Type-checking Differences Between Symbolics Common Lisp and Zetalisp

Type-checking in Zetalisp and Symbolics Common Lisp does not completely overlap for **typep** and **zl:typep**, since these two functions differ in their syntax and in the number of types each recognizes. (**typep** recognizes a much larger set of data types than **zl:typep**.)

typep accepts a type specifier in symbol or list form as its second argument, while **zl:typep** (the two-argument version) accepts a *keyword symbol* denoting a type specifier as its second argument. Since correspondences between the keyword symbols and the type specifiers are not always obvious, the list below shows the valid keywords accepted by **zl:typep** and their equivalent type specifiers accepted by **typep**. Note in particular the equivalences for **:closure**, **:fix**, **:list**, **:list-or-nil**, and **:rational**.

Zetalisp keyword (2-argument version of **zl:typep**)	Corresponds to type specifier for **typep**
:array	**array**
:atom	**atom**
:bignum	**bignum**
:closure	**dynamic-closure**
:compiled-function	**compiled-function**
:complex	**complex**
:double-float	**double-float**
:fix	**integer**
:fixnum	**fixnum**
:float	**float**
:instance	**instance**
:list	**cons**
:list-or-nil	**list**
:locative	**locative**
:non-complex-number	`(and number (not complex))`
:null	**null**
:number	**number**
:rational	**ratio**
:select-method	-----
:single-float	**single-float**
:stack-group	**sys:stack-group**
:string	**string**
:symbol	**symbol**

5.6 Declaring the Type of an Object

It is frequently useful to declare that objects should take on values of a specified type. The declaration specifiers **type** and **ftype** allow this for variable bindings and for functions. This feature is currently ignored, but is useful for programmers developing portable programs. See the section "Declarations", page 260.

6. Numbers

This chapter covers three main topics:

- types of numbers
- representation of numbers for printing and reading
- numeric functions

Zetalisp-only features, if any, are pointed out within the discussion of each topic.

Throughout this chapter, digit strings without qualifiers in running text are decimal.

6.1 Types of Numbers

Symbolics Common Lisp includes three main types of numbers: rational, floating-point, and complex. Their characteristics are described below.

6.1.1 Rational Numbers

Rational numbers are used for exact mathematical calculations. These are numbers like 0, 1, 2, -27, 15/16, -26/3, and 13/100000000000000000000. Rational numbers with no fractional part are called *integers* and those with a non-zero fractional part are called *ratios*. There is no restriction on the size of rational numbers, other than the memory available to represent them, so computations cannot "overflow" in the way they do on conventional computers.

Operations with rational numbers follow the normal rules of arithmetic and are always exact. Hence, when your program uses rational numbers, you do not have to be concerned with loss of accuracy or precision as would be the case if you used floating-point numbers.

The system automatically reduces ratios into the lowest terms. If the denominator evenly divides the numerator, Symbolics Common Lisp converts the result to an integer. This automatic reduction and conversion of ratios is called *rational canonicalization.*

```
(+ 1 1) => 2
(+ 5/6 19/3) => 43/6
(/ 1 3) => 1/3
(/ 140 -120) => -7/6
(* 12/5 10/3) => 8
(* 1000000000000 1000000000000000000) => 1000000000000000000000000000000
```

Programmers who are familiar with conventional computer systems and languages will notice that integer division in Symbolics Common Lisp is true mathematical division. The **truncate** function performs Fortran-style integer division. Other functions perform related kinds of division. See the section "Functions That Divide and Convert Quotient to Integer", page 113.

6.1.1.1 Integers

The integer data type represents mathematical integers. Symbolics Common Lisp imposes no limit on the magnitude of an integer; storage is automatically allocated as necessary to represent large integers.

Division in Zetalisp is not like mathematical division. See the section "Integer Division in Zetalisp", page 106.

Efficiency of Implementation Note

In general you need not be concerned with the details of integer representation. You simply compute in integers. Symbolics Common Lisp does, however, have two primitive types of integers, *fixnums* and *bignums*. *Fixnums* are a range of integers that the system can represent more efficiently; *bignums* are integers outside the range of fixnums.

When you compute with integers, the system represents some as fixnums and the rest (less efficiently) as bignums. The system automatically converts back and forth between fixnums and bignums based solely on the size of the integer. This automatic conversion is referred to as *integer canonicalization*.

You can ignore distinctions between fixnums and bignums in reading and printing integers. The reader uses the same syntax for fixnums and bignums, and both these types have the same printed representations.

There are a few "low level" functions that only work on fixnums, as well as some built-in system functions that require fixnums; we note this requirement in the dictionary entries for these functions.

The constants **most-negative-fixnum** and **most-positive-fixnum** give the range of fixnums on the machine. Currently in Symbolics-3600 the range is from -2147483648 to 2147483647 (-2^{31} to 2^{31}-1).

6.1.1.2 Ratios

Rational numbers that are not integers are represented as the mathematical ratio of two integers, the *numerator* and the *denominator*. The ratio is always "in lowest terms", meaning that the denominator is as small as possible. If the denominator evenly divides the numerator the system applies the rule of rational canonicalization converting the result to an integer.

The denominator is always positive; the sign of the number is carried by the numerator.

Examples:

```
6/7 => 6/7                      ;in canonical form
6/8 => 3/4                      ;converted to canonical form
-3/9 => -1/3                    ;converted to canonical form
(/ 4 -16) => -1/4               ;denominator is always positive
6/2 => 3                        ;converted to canonical form
```

6.1.2 Floating-point Numbers

Floating-point numbers are used for approximate mathematical calculations. Floating-point numbers use a restricted form of representing numbers, so that they are more efficient in some cases than rational numbers. Floating-point numbers are internally represented using a mathematical sign $s \in \{+1, -1\}$, a fraction (significand) f, and a signed exponent e. The mathematical value of the number represented is $s * f * 2^e$. The values of f and e are restricted to a certain number of (binary) digits. Symbolics Common Lisp supports two forms of floating-point numbers, corresponding to particular sizes of f and e, namely:

1. *Single-float.* Single-precision floating-point numbers have a precision of 24 bits, or about 7 decimal digits. They use 8 bits to represent the exponent. Their range is from 1.0e-45, the smallest positive denormalized single-precision number, to 3.4028235e38, the largest positive normalized single-precision number.

2. *Double-float.* Double-precision floating-point numbers have a precision of 53 bits, or about 16 decimal digits. They use 11 bits to represent the exponent. Their range is from 5.0d-324, the smallest positive denormalized double-precision floating-point number, to 1.7976931348623157d308, the largest positive normalized double-precision floating-point number.

These two forms subsume the four floating-point forms supported by Common Lisp: *Single-float* serves also as *short-float* and the system treats 1.0s0 and 1.0f0 as identical single-precision formats. Similarly, *double-float* serves also as *long-float*, with 1.0l0 and 1.0d0 treated as identical double-precision formats.

Because of the restricted representation mentioned earlier, floating-point operations don't always follow normal mathematical laws. For example,

```
(1.0e10 + -1.0e10) + 1.0 ≠ 1.0e10 + (-1.0e10 + 1.0)
```

Floating-point is appropriate for situations where there is no exact rational answer to a problem (for instance **pi**, **(sqrt 2)**), or where exact answers are not required. Because of the approximate nature of floating-point numbers and calculations, seemingly anomalous behavior can occur. (This observation holds in general, regardless of any particular implementation.)

Examples:

```
(- 6 5.9) => 0.099999905
(- 2 1.9) => 0.100000024
(- 2 1.9d0) => 0.10000000000000009d0
(- 1000000.1d0 1000000) => 0.09999999997671694d0
(- 100000.1d0 100000) => 0.10000000000582077d0
(* .001 10) => 0.010000001
(* .0003d0 10) => 0.0029999999999999996d0
(/ 1.0 3) => 0.33333334
(/ 1.0d0 3) => 0.3333333333333333d0
(/ 1.0 6) => 0.16666667
(/ 1.0d0 6) => 0.16666666666666666d0
```

This is all "correct" and is due to interactions between the restricted representation of floating-point numbers and the problems of changing bases from binary (internal representation) to decimal (printed representation).

Floating-point computations can get exponent overflow or underflow, if the result is too large or small to be represented. Exponent overflow always signals an error. Exponent underflow normally signals an error, unless the computation is inside the body of a **without-floating-underflow-traps**. Any time a floating-point error occurs, you are offered a way to proceed from it, by substituting the IEEE floating-point result for the mathematical result.

Example:

```
(* 4s-20 4s-20)          ;evaluating this signals an error
(without-floating-underflow-traps  (* 4s-20 4s-20)) => 1.6e-39
```

The constants below indicate the range for single- and double-floating-point numbers. Constants for short- and long-floating point formats appear in the Dictionary of Numeric Functions and Variables; these constants have the same values as single- and double-floating point formats respectively.

Floating-point Efficiency Note

Single-precision floating-point is significantly more efficient than double-precision floating-point. In particular, double-precision numbers take up more memory than single-precision numbers.

6.1.2.1 Constants Indicating the Range of Floating-point Numbers

Constant	Value
most-positive-single-float	`3.4028235e38`
least-positive-single-float	`1.1754944e-38`
least-negative-single-float	`-1.1754944e-38`
most-negative-single-float	`-3.4028235e38`
most-positive-double-float	`1.7976931348623157d308`
least-positive-double-float	`2.2250738585072014d-308`
least-negative-double-float	`-2.2250738585072014d-308`
most-negative-double-float	`-1.7976931348623157d308`

6.1.2.2 IEEE Floating-point Representation

The Symbolics Lisp Machine uses IEEE-standard formats for single-precision and double-precision floating-point numbers. Number objects exist that are outside the upper and lower limits of the ranges for single and double precision. Larger than the largest number is +1e∞ (or +1d∞ for doubles). Smaller than the smallest number is -1e∞ (or -1d∞ for doubles). Smaller than the smallest normalized positive number but larger than zero are the "denormalized" numbers. Some floating-point objects are Not-a-Number (NaN); they are the result of (/ 0.0 0.0) (with trapping disabled) and like operations.

IEEE numbers are symmetric about zero, so the negative of every representable number is also a representable number. Zeros are signed in IEEE format, but +0.0 and -0.0 act the same arithmetically as 0.0. However, they are distinguishable to non-numeric functions. For example:

```
(= +0.0 -0.0) => t
(minusp -0.0)  => NIL
(plusp 0.0)    => nil
(plusp -0.0)   => nil
(zerop -0.0)   => t
(eql 0.0 -0.0) => nil
```

See, "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, *An American National Standard*, August 12, 1985.

See the section "Numeric Type Conversions", page 112.

6.1.3 Complex Numbers

A complex number is a pair of noncomplex numbers, representing the real and imaginary parts of the number. The real and imaginary parts can be rational, single-float, or double-float, but both parts always have the same type. Hence we distinguish between complex rational and complex floating-point numbers.

In Symbolics Common Lisp a complex rational number can never have a zero imaginary part. The system matches up the real and imaginary parts of a complex number operand or result; if the real part is rational and the imaginary part is a zero integer, the system converts the complex number to a noncomplex rational number. This matching of types and conversion is called the rule of *complex canonicalization.*

Conversion does not occur if the result is a complex floating-point number with a zero imaginary part. For example #C(5.0 0.0) is not automatically converted to 5.0. In this case if you want to convert to a noncomplex number you must call the appropriate conversion function. See the section "Numeric Type Conversions", page 112.

Complex numbers are used when mathematically appropriate.

```
(sqrt -1) => #C(0 1)
(log -1) => #C(0.0 3.1415927)
(+ #C(4 10) #C(5 -10)) => 9
(+ #C(4.0 10) #C(5.0 -10)) => #C(9.0 0.0)
```

Note: In Zetalisp, the functions **sqrt** and **log** would signal an error if given a negative argument, instead of returning a complex number as they do in the previous examples.

6.1.4 Type Specifiers and Type Hierarchy for Numbers

The type specifiers relating to numeric data types are as follows:

number	integer	short-float
rational	ratio	long-float
float	single-float	fixnum
complex	double-float	signed-byte
bignum	unsigned-byte	bit

Details about each type specifier appear in its dictionary entry.

Figure 2 shows the relationships between numeric data types. For more on data types, type specifiers, and type checking in Symbolics Common Lisp: See the section "Data Types and Type Specifiers", page 69.

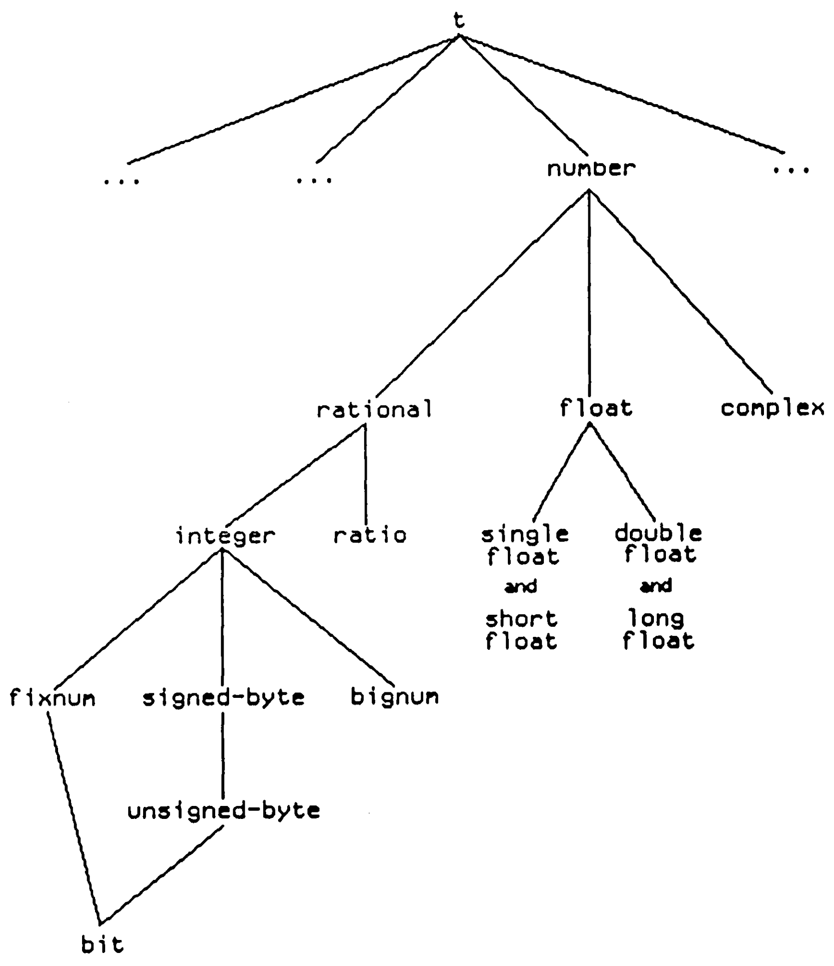


Figure 2. Symbolics Common Lisp Numeric Data Types

6.2 How the Printer Deals with Numbers

Numbers can be printed in a variety of ways as determined by the value of control variables.

'Escape' characters such as the backslash (or slash in Zetalisp) do not affect the printing of numbers.

6.2.1 Printed Representation of Rational Numbers

Rational numbers can print in any radix between 2 and 36 (inclusive), depending on the value you assign to the control variable ***print-base***. The default value is 10. (Zetalisp uses the value of **zl:base** to control printing.)

When ***print-base*** has a value over 10, digits greater than 9 are represented by means of alphabetical characters.

If an integer is negative, a minus sign is printed, followed by the absolute value of the integer. The integer zero is represented by the single digit 0 and never has a sign. Integers in base ten print with or without a trailing decimal point, depending on the value of ***print-radix***. See the section "Radix Specifiers for Rational Numbers", page 90.

To allow printing of integers in other than Arabic notation, ***print-base*** can be set to a symbol that has a **si:princ-function** property (such as **:english** or **:roman**). The value of the property is applied to two arguments:

- - of the number to be printed
- The stream to which to print it

The printer prints ratios as follows:

- a minus sign if the ratio is negative
- the absolute value of the numerator
- a slash (/) character (Zetalisp uses a backslash, \)
- the denominator

Ratios print in canonical form.

6.2.1.1 Radix Specifiers for Rational Numbers

You can specify that a *radix specifier* be used to show in what radix a number is being printed. To do so, set the control variable ***print-radix*** to **t** (default value is **nil**). The radix specifier is always printed with a lowercase letter.

Radix Specifier Format

The general format of a radix specifier is a sequence of the following characters:

- #

- a non-empty sequence of decimal digits representing an unsigned decimal integer *n* (must be in the range 2 - 36 inclusive)
- r

immediately followed by:

- an optional sign
- a sequence of digits in radix *n*

There are special abbreviations for commonly used radices such as binary, octal, and hexadecimal.

Radix	Normal Form	Abbreviated Form	Uppercase Form (Reader only)
Binary	#2r	#b	#2R or #B
Octal	#8r	#o	#8R or #O
Hexadecimal	#16r	#x	#16R or #X

For integers in base ten the radix specifier uses a trailing decimal point instead of a leading radix specifier. When ***print-radix*** is set to **nil**, integers in base ten are printed without a trailing decimal point.

To print a ratio with a radix specifier, the printer uses the same notation as for integers, except in the case of decimals. Ratios in decimal are printed using the #10r notation.

Examples (integers):

```
(+ 2 3) => 5
(setq *print-base* 2) => 10
(+ 2 3) => 101
(setq *print-radix* t) => T
(+ 2 3) => #b101
(setq *print-base* 16) => #x10
(* 6 2) => #xC
(setq *print-base* 10) => 10.
(* 5 8) => 40.
(setq *print-radix* nil) => NIL
(* 5 8) => 40
(setq *print-base* ':roman)  => :ROMAN
(* 5 8)  => XL
```

Examples (ratios):

```
4/5 => 4/5
(setq *print-radix* t) => T
4/5 => #10r4/5
(setq *print-base* 8) => #o10
4/12 => #o1/3
5/9 => #o5/11
(setq *print-base* 5) => #5r10
7/30 => #5r12/110
```

6.2.2 Printed Representation of Floating-point Numbers

Floating-point numbers are always printed in decimal. For a single-precision floating-point number, the printer first decides whether to use ordinary notation or exponential notation. If the magnitude of the number is so large or small that the ordinary notation would require an unreasonable number of leading or trailing zeroes, exponential notation is used. A floating-point number is printed as follows:

- An optional leading minus sign
- One or more digits
- A decimal point
- One or more digits
- Optionally an *exponent marker*, described below, an optional minus sign, and the power of ten

The exponent marker (also referred to as the exponent character or letter) indicates the number's floating-point format. The printer uses one of the following characters: `s`, `f`, `l`, `d`, or `e`. These indicate short, single, long, and double floating point numbers respectively. `e` indicates a number format which corresponds to the current value of variable ***read-default-float-format***. This variable takes a value denoting one of the valid floating-point formats, namely **short-float, single-float, long-float,** and **double-float.**

To decide whether to print an exponent marker and if so of which type, the printer checks the value of ***read-default-float-format*** and applies the rules summarized below.

Notation used	Does number's format match current value of ***read-default-float-format***?	Exponent marker
Ordinary	Yes	Don't print marker
	No	Print marker and zero
Exponential	Yes	Print e
	No	Print marker

Examples:

```
(setq *read-default-float-format* 'single-float) => SINGLE-FLOAT
1.0s0 => 1.0
1.0s7 => 1.0e7
1.0d0 => 1.0d0
1.0d7 => 1.0d7
(setq *read-default-float-format* 'double-float) => DOUBLE-FLOAT
1.0s0 => 1.0f0
1.0s7 => 1.0f7
1.0d0 => 1.0
1.0d7 => 1.0e7
```

The number of digits printed is the "correct" number; no information present in the number is lost, and no extra trailing digits are printed that do not represent information in the number. Feeding the printed representation of a floating-point number back to the reader is always supposed to produce an equal floating-point number.

The printed representation for floating-point "infinity" is as follows:

- A plus or minus sign
- The digit "1"
- The appropriate exponent mark character
- An infinity sign: ∞

Examples:

```
(setq *read-default-float-format* 'double-float) => DOUBLE-FLOAT
+1s∞ => +1f∞
+1d∞ => +1e∞
(setq *read-default-float-format* 'single-float) => SINGLE-FLOAT
-1s∞ => -1e∞
-1l∞ => -1d∞
```

6.2.2.1 Control Variables for Printing Numbers

print-base	Specifies radix for printing numbers (range 2-36, default 10)
print-radix	Determines the printing or suppression of radix specifier (value **t** or **nil**)
read-default-float-format	Guides the printer in choice of exponent marker for floating-point number
zl:base	Another name for ***print-base***

6.2.3 Printed Representation of Complex Numbers

The printed representation for complex numbers is:

`#C(`*realpart imagpart*`)`

The real and imaginary parts of the complex number are printed in the manner appropriate to their type.

Examples:

```
(+ #C(3.4 5) 6) => #C(9.4 5.0)
(* 4 #C(2.0d0 5)) => #C(8.0d0 20.0d0)


(setq *print-radix* t)
(setq *print-base* 16)
(+ #C(3 4) #C(8 9)) => #C(#xB #xD)
```

6.3 How the Reader Recognizes Numbers

The Symbolics Common Lisp reader accepts characters, accumulates them into a token, and then interprets the token as a number or a symbol. In general, the token is interpreted as a number if it satisfies the syntax for numbers. Often, the interpretation is determined by the value of control variables, as explained below.

6.3.1 How the Reader Recognizes Rational Numbers

The reader determines the radix in which integers and ratios are to be read in the following manner:

- If the number is preceded by a radix specifier, the reader interprets the rational number using the specified radix.
- If the number is an integer with a trailing decimal point, the reader uses a radix of ten.
- In the absence of a radix specifier, or a trailing decimal point for integers, the reader determines the radix by checking the current value of the control variable ***read-base***. (Zetalisp uses the value of **zl:ibase**.)
- The reader accepts radix specifier syntax in both upper and lowercase characters. See the section "Radix Specifier Format", page 90.

Examples:

```
(+ #2r101 #2r11) => 8
(+ #3r11 #5r101) => 30
(* #b100 #xC) => 48
(* #o15 #8r5) => 65
(* #b11/10 40) => 60            ;*read-base* is 10
(setq *read-base* 2) => 2
(+ 100 1101) => 17
(* #x10/a 101) => 8
```

6.3.1.1 How the Reader Recognizes Integers

The syntax for a simple integer is:

- An optional plus or minus sign
- A string of digits
- An optional decimal point

If the trailing decimal point is present, the digits are interpreted in decimal radix; otherwise, they are considered as a number whose radix is the value of the variable ***read-base*** (or **zl:ibase** in Zetalisp). Valid values are between 2 and 36, inclusive, default value is 10.

read understands simple integers as well as a simple integer followed by an underscore (_) or a circumflex (^), followed by another simple integer. The two simple integers are interpreted in the usual way and the character between them indicates an operation to be performed on the two integers.

- The underscore indicates a binary "left shift"; that is, the integer to its left is doubled the number of times indicated by the integer to its right. For example, **645_6** means **64500** (in octal).
- The circumflex multiplies the integer to its left by ***read-base*** the number of times indicated by the integer to its right. (The second integer is not allowed to have a leading minus sign.) For example, **645^3** means **645000.**

Here are some examples of valid representations of integers to be given to **read**:

```
4  => 4
23456.  => 23456
-546  => -546
+45^+6  => 45000000
2_11   => 4096
```

Compatibility Note

In Symbolics Common Lisp, setting the value of ***read-base*** at greater than ten does not cause the reader to interpret tokens as numbers rather than symbols. Tokens with names such as **a**, **b**, and **face** are still interpreted as symbols rather than numbers, unless the value of control variables **si:*read-extended-ibase-signed-number*** and **si:*read-extended-ibase-unsigned-number*** is set to **t**.

See the section "Reading Integers in Bases Greater Than 10", page 96.

This is an incompatible difference from the language specification presented in Guy L. Steele's *Common Lisp, the Language.*

Reading Integers in Bases Greater Than 10

The reader uses letters to represent digits greater than 10. Thus, when ***read-base*** is greater than 10, some tokens could be read as either integers, floating-point numbers, or symbols. The reader's action on such ambiguous tokens is determined by the value of **si:*read-extended-ibase-unsigned-number*** and **si:*read-extended-ibase-signed-number***. Setting these variables to **t** causes the tokens to be always interpreted as numbers. A **nil** setting causes the tokens to be interpreted as symbols or floating-point numbers. The above variables can have two other intermediate settings as explained in the Dictionary entry.

Examples:

```
(setq *read-base* 16) => 16
(+ 10 5) => 21                  ;this works as expected
(+ c 2) => signals an error because c is an unbound symbol
(setq si:*read-extended-ibase-signed-number* t) => T
(+ c 2) => 14                   ;now c is read as a number
(+ #xc 2) => 14         ;always safe
```

6.3.1.2 How the Reader Recognizes Ratios

The syntax of a ratio is

- An optional sign
- A string of digits
- A / (slash character)
- A string of digits

The Zetalisp syntax is identical, except that a *backslash* character (\), is used instead of a slash.

A ratio can also be prefixed by a radix specifier. See the section "Radix Specifiers for Rational Numbers", page 90.

Ratios written with a radix specifier are read in the radix specified. Ratios written without a radix specifier are always read in the current ***read-base*** (or **zl:ibase** in Zetalisp).

Examples:

```
-14/32 => -7/16
22/7 => 22/7
#o24/17 => 4/3          ;20/15 => 4/3
#x4f/10 => 79/16
(setq *read-base* 2) => 2
101/10 => 5/2
 #10r101/10 => 101/10
```

6.3.2 How the Reader Recognizes Floating-point Numbers

Floating-point numbers are always read in decimal radix.

The syntax for floating-point numbers has two possible formats:

- An optional plus or minus sign
- (Optionally) some digits
- A decimal point
- One or more digits

- An optional exponent marker, consisting of an exponent letter, an optional minus sign, and digits representing the power of ten

or

- An optional sign
- A string of digits
- Optionally a decimal point followed by optional digits
- An exponent marker

Note that in the first format a decimal point is mandatory, but the exponent marker is optional. In the second representation the decimal point can be omitted, but the exponent marker is always present. Moreover, the optional sign is always followed by at least one digit.

Here are some examples of floating-point numbers in both formats:

```
Format 1        Format 2

20.2e-4         20.2e-4
.202e-2         202.e-5
.00202          202e-5
```

If no exponent is present, the number is a single-float. If an exponent is present, the exponent letter determines the type of the number.

6.3.2.1 Floating-point Exponent Characters

Following is a summary of floating-point exponent characters and the way numbers containing them are read.

Character	*Floating-point precision*
D or d	double-precision
E or e	depends on value of ***read-default-float-format***
F or f	single-precision
L or l	double-precision
S or s	single-precision

The variable ***read-default-float-format*** controls how floating-point numbers with no exponent or an exponent preceded by "E" or "e" are read. Here is a summary of the way possible values cause these numbers to be read.

Value	*Floating-point precision*
single-float	single-precision
short-float	single-precision
double-float	double-precision
long-float	double-precision

The default value is **single-float.**

As a special case, the reader recognizes IEEE floating-point "infinity". The syntax for infinity is as follows:

- A required plus or minus sign
- The digit "1"
- Any of the exponent mark characters
- And the exponent character, which must be an infinity sign: ∞

Here are some examples of printed representations that read as single-floats:

```
0.0  => 0.0
1.5  => 1.5
14.0  => 14.0
0.01 => 0.01
.707  => 0.707
-.3  => -0.3
+3.14159  => 3.14159
6.03e23  => 6.03e23     ;only when *read-default-float-format*
1E-9  => 1.0e-9         ;  is 'single-float
1.e3  => 1000.0
+1e∞   => +1e∞
(setq *read-default-float-format* 'double-float) => DOUBLE-FLOAT
3.14159s0  => 3.14159
1.6s-19  => 1.6e-19
```

Here are some examples of printed representations that read as double-floats (current value of ***read-default-float-format*** is **single-float**).

```
0d0    => 0.0d0
1.5d9  => 1.5d9
-42D3  => -42000.0d0
1.d5  => 100000.0d0
-1d∞    => -1d∞
(setq *read-default-float-format* 'double-float) => DOUBLE-FLOAT
0.0  => 0.0
6.03e23  => 6.03e23
-1e∞  => -1e∞
```

6.3.2.2 Control Variables for Reading Numbers

read-base	Holds radix for reading of rational numbers (2-36, default 10)
read-default-float-format	Controls reading of floating-point number with no exponent or exponent e (or E)
zl:ibase	Another name for ***read-base***
si:*read-extended-ibase-unsigned-number*	Controls reading of unsigned integers in bases greater than ten
si:*read-extended-ibase-signed-number*	Controls reading of signed integers in bases greater than ten

6.3.3 How the Reader Recognizes Complex Numbers

The reader recognizes #C(*number1 number2*) as a complex number. *number1* and *number2* can be of any noncomplex type (coercion is applied if necessary). *number1* is used as the real part and *number2* is used as the imaginary part. The resulting Lisp object is represented in complex canonical form.

Examples:

```
#C(3.0 4.0) ==> #C(3.0 4.0)
#C(1 0) ==> 1
#C(1/2 3) ==> #C(1/2 3)
#C(1/2 3.0) ==> #C(0.5 3.0)
```

6.4 Numeric Functions

As stated earlier, most numeric functions in Symbolics Common Lisp are generic, rather than applicable to a specific number type. Generic functions include predicates which check numeric type and properties, functions which perform numeric comparison and arithmetic, functions allowing numeric data conversions, transcendental functions, and a pseudo-random number generator facility. There is also a group of functions which do machine-dependent arithmetic.

Two groups of functions work for integers only. One group performs logical operations on integers, the other is a group of byte-manipulation functions. For purposes of logical and byte manipulation operations an integer is treated as a sequence of bits, with the low bit in the rightmost position. The byte-manipulation functions let you operate on any number of contiguous bits within the integer.

6.4.1 Coercion Rules for Numbers

When arithmetic and numeric comparison functions of more than one argument receive arguments of different numeric types, these must be converted to a common type. Symbolics Common Lisp does the conversion by following uniform *coercion rules*. For functions returning a number, the coerced argument type is also the type of the result. Note: The functions **max** and **min** are two notable exceptions where no conversion is performed.

Here are the coercion rules for the different argument types.

Argument Types		Converted to	Result Type
Single-float	Rational	Single-float	Single-float
Double-float	Rational	Double-float	Double-float
Single-float	Double-float	Double-float	Double-float
Complex	Non-complex	Complex	Complex

Since rational computations are always exact, you need not be concerned with coercions among rational number types.

Since floating-point computations with different precisions can lead to inefficiency, inaccuracy, or unexpected results, Symbolics Common Lisp does not automatically convert between double-floats and single-floats if all the arguments are of the same floating-point type.

Thus, if the constants in a numerical algorithm are written as single-floats (assuming this provides adequate precision), and if the input is a single-float, the computation is done in single-float mode and the result is a single-float. If the input is a double-float the computations are done in double precision and the result is a double-float, although the constants still have only single precision. For most algorithms, it is desirable to have two separate sets of constants to maintain accuracy for double precision and speed for single precision.

This means that a single-float computation can get an exponent overflow error even when the result could have been represented as a double-float. For example, 1.0e18 * 1.0e22 would create an exponent overflow error, even though the result could be represented by the valid double-float number 1.0d40. You can prevent this type of error by converting one, or both of the arguments to a larger data type.

In general then, floating-point number computations yield a floating-point result of the type corresponding to the largest floating-point type in the argument list. Computations with rational numbers yield a rational number result.

Examples:

```
(+ 1 3.0) => 4.0
(+ 2 4d0) => 6.0d0
(+ 3s0 4d0) => 7.0d0
(+ #C(6 8) 2) => #C(8 8)
(+ #C(4 9) 7.0d1) => #C(74.0d0 9.0d0)
(+ #C(3.4s5 9.2s3) #C(6.2d4 0.8d4)) => #C(402000.0d0 17200.0d0)
(+ #C(4 -3) #C(6 3)) => 10
(+ #C(3.0 8.0) #C(4.5 -8.0)) => #C(7.5 0.0)
```

6.4.2 Numeric Function Categories

The following groups of numeric functions are available

- Numeric Predicates
- Numeric Comparisons
- Arithmetic Functions
- Transcendental Functions
- Numeric Type Conversions
- Logical Operations on Numbers
- Byte Manipulation Functions
- Random Number Generation
- Machine-dependent Arithmetic Functions

The discussion for each function group is followed by a summary table of the functions in that category. The alphabetized Dictionary provides complete coverage of each individual function. See the document *Symbolics Common Lisp: Language Dictionary*.

6.4.3 Numeric Predicates

Numeric predicates test the data types of numbers, as well as some numeric properties such as whether the number is odd or even. The summary tables below group numeric predicates by function.

6.4.3.1 Numeric Type-checking Predicates

These predicates test a number to see if it belongs to a given type. General type-checking functions such as **typep** and **subtypep** can also be used to determine relationships within the hierarchy of numeric types and for similar purposes. For more on these functions: See the section "Determining the Type of an Object", page 79.

complexp *object*	Tests for complex number.
floatp *object*	Tests for floating-point number of any precision.
integerp *object*	Tests for integer.
numberp *object*	Tests for number of any type.
rationalp *object*	Tests for rational number.
sys:double-float-p *object*	Tests for double-precision floating-point number.
sys:single-float-p *object*	Tests for single-precision floating-point number.
zl:bigp *object*	Tests for bignum.
zl:fixnump *object*	Tests for fixnum.
zl:fixp *object*	Tests for integer (same as **integerp**).
zl:flonump *object*	Tests for single-precision floating-point number (same as **sys:single-float-p**).
zl:rationalp *object*	Tests for ratio.

6.4.3.2 Numeric Property-checking Predicates

evenp *integer*	tests for even integers
oddp *integer*	tests for odd integers
minusp *number*	tests if number is less than zero
plusp *number*	tests if number is greater than zero
zerop *number*	tests if a number is zero
zl:signp *test number*	tests if the sign of *number* is *test*

6.4.4 Numeric Comparisons

Symbolics Common Lisp supports eight numeric comparison functions in which the values of two or more arguments are compared with respect to equality, inequality, less-than, greater-than, and so on.

All of these functions require that their arguments be numbers, and signal an error if given a nonnumber. They work on all types of numbers, automatically performing any required coercions. Note that no coercion takes place for functions **max** and **min**.

= and ≠ work for complex number comparisons. All other comparison functions require non-complex numbers as arguments.

6.4.4.1 Types of Equality

In general we can distinguish two types of equality:

- Equality of two Lisp objects, tested by predicates **eq**, **eql**, **equal**, and **equalp**. See the section "Comparison-performing Predicates", page 282.

- Numeric equality, tested by =.

Although predicates **eq**, **eql**, **equal**, and **equalp** can take numbers as arguments, they cannot be used interchangeably with =, because they don't work the same way:

- **eq** produces unreliable results on numbers.

- **eql**, **equal** and **equalp** are true for numeric arguments of the same numeric value *and* type. (No coercion is performed.) In addition, floating-point zeros of differing signs do not satisfy any of these predicates.

- = takes only numbers as arguments; it is true if its arguments are of the same numeric value, *regardless of type*. Floating-point zeros are = to any other zero values, regardless of sign.

For comparing numeric values, = is therefore the preferred function.

Examples:

```
(eql 3 3.0) => NIL
(= 3 3.0) => T

(eq 10.0d0 (* 5.0d0 2)) => NIL
(= 10.0d0 (* 5.0d0 2))  => T

(equal #C(5.0 0) 5.0) => NIL
(= #C(5.0 0) 5.0) => T

(eql 0.0 -0.0)  => NIL
(= 0.0 -0.0)  => T

(= 3 nil)   ;generates an error
(eql 3 nil)  => NIL
```

The following function can return either **t** or **nil**.

```
(defun foo ()
   (let ((x (float 5)))
     (eq x (car (cons x nil)))))
```

6.4.4.2 Numeric Comparison Functions

Function	*Synonyms*	*Comparison/Returned Value*
≠ *number* &rest *numbers*	/=	Not equal
< *number* &rest *more-numbers*	**zl:lessp**	Less-than
≤ *number* &rest *more-numbers*	<=	Less-than-or-equal
= *number* &rest *more-numbers*		Equal
> *number* &rest *more-numbers*	**zl:greaterp**	Greater-than
≥ *number* &rest *more-numbers*	>=	Greater-than-or-equal
max *number* &rest *more-numbers*		Greatest of its arguments
min *number* &rest *more-numbers*		Least of its arguments

6.4.5 Arithmetic

All of these functions require that their arguments be numbers, and signal an error if given a nonnumber. With a few exceptions that require integer arguments, arithmetic functions work on all types of numbers, automatically performing any required coercions. See the section "Coercion Rules for Numbers", page 101.

6.4.5.1 Integer Division in Zetalisp

Integer division in Zetalisp returns an integer rather than the exact rational-number result. The quotient is truncated toward zero rather than rounded. The exact rule is that if A is divided by B, yielding a quotient of C and a remainder of D, then $A = B * C + D$ exactly. D is either zero or the same sign as A. Thus the absolute value of C is less than or equal to the true quotient of the absolute values of A and B. This is compatible with Maclisp and conventional computer hardware. However, it has the serious problem that it does *not* obey the rule that if A divided by B yields a quotient of C and a remainder of D, then dividing $A + k * B$ by B yields a quotient of $C + k$ and a remainder of D for all integer k. The lack of this property sometimes makes Zetalisp integer division hard to use. For a more detailed discussion of truncation and rounding off operations: See the section "Numeric Type Conversions", page 112.

6.4.5.2 Arithmetic Functions

Zetalisp functions appear immediately under their Common Lisp synonyms.

Function	*Action*
+ &rest *numbers* **zl:+$** **zl:plus**	Returns the sum of its arguments.
- *number* &rest *more-numbers* **zl:-$**	First argument minus the sum of the rest of the arguments, or negative of single argument.
abs *number*	Returns the absolute value of *number*.
conjugate *number*	Returns the complex conjugate of *number*.
* &rest *numbers* **zl:*$** **zl:times**	Returns the product of its arguments.
/ *number* &rest *more-numbers*	Returns the first argument divided by the product of the rest of the arguments, or reciprocal of single argument. Returns integer or ratio, as appropriate, when arguments are rational.
1+ *number* **zl:1+$** **zl:add1**	Adds 1 to *number*.
1- *number* **zl:1-$** **zl:sub1**	Subtracts 1 from *number*

gcd &rest *integers*	Greatest common divisor of all its arguments.
lcm &rest *integers*	Least common multiple
rem *number divisor* **zl:\** **zl:remainder**	Returns remainder of *number* divided by *divisor.*
mod *number divisor*	Performs **floor** on its arguments (divides *number* by *divisor*, truncating result toward negative infinity), but only returns the second result of **floor**, the remainder.
expt *base-number power-number*	Returns *base-number* raised to the power *power-number.*
sqrt *number*	Returns the square root of *number.*
isqrt *integer*	Returns the greatest integer less than or equal to the square root of its argument.
signum *number*	If *number* is rational, returns -1, 0, or 1, to indicate that the argument is negative, zero or positive. Floating-point and complex arguments produce different results.
zl:difference *number* &rest *more-numbers*	Returns the first argument minus the sum of the rest of the arguments.
zl:minus *number*	Returns the negative of *number.*
zl:/ *number* &rest *more-numbers* **zl:/$**	Returns the first argument divided by the product of the rest of the arguments, or

	reciprocal of single argument. Truncates results for integer arguments.
zl:quotient *number* &rest *more-numbers*	With more than one argument, same as **zl:/**; with single argument, returns the reciprocal of *number*; truncates result for integer arguments.
zl:gcd *integer1 integer2* &rest *more-integers* **zl:\\**	Greatest common divisor of all its arguments.
zl:expt *num expt* **zl:^** **zl:^$**	Returns *num* raised to the *expt*th power.
zl:sqrt *n*	Returns the square root of *n*.

6.4.6 Transcendental Functions

This group includes four subsets of transcendental functions: powers of *e* (where *e* is the base of natural logarithms), logarithmic functions, trigonometric and related functions, and hyperbolic functions.

These functions are only for floating-point arguments; if given an integer they convert it to a single-float and return a single-float. If given a double-float, they return a double-float.

6.4.6.1 Powers Of e and Log Functions

exp *number*	Returns e raised to power *number*
log *number* &optional *base*	Returns the natural logarithm of *number*, or optionally, the logarithm of *number* in the base *base*
zl:log *x*	Returns the natural logarithm of *x*

6.4.6.2 Trigonometric and Related Functions

sin *radians*	Returns the sine of its argument.
cos *radians*	Returns the cosine of its argument.
tan *radians*	Returns the tangent of its argument.
tand *degrees*	Returns the tangent of *degrees.*
sind *degrees*	Returns the sine of *degrees.*
cosd *degrees*	Returns the cosine of *degrees.*
cis *radians*	Returns (**complex** (**cos** *radians*) (**sin** *radians*)).
asin *number*	Returns the arc sine of *number* in radians.
acos *number*	Returns the arc cosine of *number* in radians.
atan *y* &optional *x*	Returns the angle between -π and π radians whose tangent is *y/x.*
phase *number*	Returns the angle part (in radians) of the polar representation of a complex number. (The function **abs** returns the radius part of the complex number.)
zl:atan *y x*	Returns angle between 0 and 2π in radians.
zl:atan2 *y x*	Returns angle in radians (same as **atan**).

6.4.6.3 Hyperbolic Functions

sinh *number*	Returns the hyperbolic sine of *number*, where *number* is in radians
cosh *number*	Returns the hyperbolic cosine of *number*, where *number* is in radians
tanh *number*	Returns the hyperbolic tangent of *number*, where *number* is in radians
asinh *number*	Returns the hyperbolic arc sine of *number*.
acosh *number*	Returns the hyperbolic arc cosine of *number*
atanh *number*	Returns the hyperbolic arc tangent of *number*

6.4.7 Numeric Type Conversions

These functions are provided to allow specific conversions of data types to be forced when desired. When converting to an integer, you can select any of the following:

- truncation toward negative infinity (**floor**, **ffloor**, **zl:fix**)
- truncation toward positive infinity (**ceiling**, **fceiling**)
- truncation toward zero (**truncate**, **ftruncate**)
- rounding to the nearest integer (**round**, **fround**, **zl:fixr**)

See the section "Comparison Of **floor**, **ceiling**, **truncate** And **round**", page 114.

In addition, these functions extract specific components of ratios, floating-point, and complex numbers such as the denominator of a ratio, or the imaginary part of a complex number.

6.4.7.1 Functions That Convert Noncomplex to Rational Numbers

rational *number*	Converts a noncomplex number to an equivalent rational number
rationalize *number*	Converts a noncomplex number to a rational number in best available approximation of its format
zl:rational *x*	Converts a noncomplex number to an equivalent rational number (same as **rationalize**)

6.4.7.2 Functions That Convert Numbers to Floating-point Numbers

float *number* &optional *other*	Converts *number* to floating point with the precision of *other*; with single argument, converts *number* (if non-floating) to single-precision floating point, else to double-precision.
zl:dfloat *x*	Converts a number to double-precision floating-point number
zl:float *x*	Converts a number to a single-precision floating-point number

6.4.7.3 Functions That Divide and Convert Quotient to Integer

floor *number* &optional (*divisor* **1**)	Divides *number* by *divisor*, truncates result toward negative infinity*
ceiling *number* &optional (*divisor* **1**)	Divides *number* by *divisor*, truncates result toward positive infinity*
truncate *number* &optional (*divisor* **1**)	Divides *number* by *divisor*, truncates result toward zero*
round *number* &optional (*divisor* **1**)	Divides *number* by *divisor*, rounds the result*
zl:fix *x*	Converts *x* from a floating-point number to an integer by truncating (similar to **floor**)
zl:fixr *x*	Converts *x* from a floating-point number to an integer by rounding (similar to **round**)

*See the section "Comparison Of **floor**, **ceiling**, **truncate** And **round**", page 114.

6.4.7.4 Functions That Divide and Return Quotient as Floating-point Number

ffloor *number* &optional (*divisor* **1**)	Like **floor**, except result is a floating-point number instead of an integer
fceiling *number* &optional (*divisor* **1**)	Like **ceiling**, except result is a floating-point number instead of an integer
ftruncate *number* &optional (*divisor* **1**)	Like **truncate**,except result is floating-point number instead of integer
fround *number* &optional (*divisor* **1**)	Like **round**, except result is a floating-point number

6.4.7.5 Functions That Extract Components From a Rational Number

numerator *rational*	If the argument is a ratio, returns the numerator of *rational*. For an integer argument, returns *rational*
denominator *rational*	If the argument is a ratio, returns denominator of *rational*. For an integer argument, returns 1

6.4.7.6 Functions That Decompose and Construct Floating-point Numbers

decode-float *float*	Returns values representing the significand, the exponent, and the sign of the argument
integer-decode-float *float*	Similar to **decode-float** except it scales the significand so as to be an integer
float-digits *float*	Returns the number of radix digits used in the representation of the argument
float-precision *float*	Returns the number of significant radix digits in the argument
float-radix *float*	Returns integer radix of floating-point argument
float-sign *float1* &optional *float2*	Returns a floating-point number of the same sign as *float1* and of the same absolute value as *float2*; *float2* defaults to a floating-point of the same precision as *float1*
scale-float *float integer*	Returns (*float* * $2^{integer}$)

6.4.7.7 Functions That Decompose and Construct Complex Numbers

complex *realpart* &optional *imagpart*	Builds a complex number from real and imaginary noncomplex parts.
realpart *number*	If *number* is complex, returns the real part of *number*. If *number* is noncomplex, returns *number*.
imagpart *number*	If *number* is complex, returns its imaginary part. If *number* is noncomplex, returns zero of the same type as *number*.

6.4.7.8 Comparison Of floor, ceiling, truncate And round

floor, ceiling, truncate, and **round** all produce two values. The second result, the remainder, is omitted from the table below.

```
(floor 1.8) => 1 and 0.79999995
(floor -1.8) => -2 and 0.20000005
(floor 5 3) => 1 and 2
(ceiling 5 3) => 2 and -1
(truncate 5 3) => 1 and 2
(round 5 3) => 2 and -1
(round -5 3) => -2 and 1
(round 5 -3) => -2 and -1
```

Argument	**floor**	**ceiling**	**truncate**	**round**
1.8	1	2	1	2
1.5	1	2	1	2
1.3	1	2	1	1
0.9	0	1	0	1
0.5	0	1	0	0
0.2	0	1	0	0
-0.2	-1	0	0	0
-0.9	-1	0	0	-1
-1.3	-2	-1	-1	-1
-1.5	-2	-1	-1	-2
-1.8	-2	-1	-1	-2

6.4.8 Logical Operations on Numbers

The logical functions described here are bit-wise operations which require integers as arguments; a non-integer argument signals an error. Logical operations on integers operate on the internal binary representation of the integer. Moreover, integers are treated as though they were "sign-extended". That is, negative integers have all one-bits on the left, and non-negative integers have all 0-bits on the left. As described below, this provides a convenient way of representing infinite vectors of bits as well as sets.

The functions fall into three main logical groupings: those which perform a specified bit-wise logical operation on their arguments and return the result, those which return specific components or characteristics of their argument, and a group of predicates based on bit-testing.

6.4.8.1 Infinite Bit-vectors and Sets Represented by Integers

It is noteworthy that these logical operations can be applied to infinite bit-vectors, if these are represented by integers. This applies to infinite bit-vectors in which only a finite number of bits are one, or only a finite number of bits are zero.

Suppose that the bits in such a vector are indexed by the non-negative integers j=0,1,..., and that bit j is assigned a "weight" 2^j. Then, a vector with only a finite number of one-bits is represented by the positive integer corresponding to the sum of the weights of the one-bits. Similarly, a bit-vector with only a finite number of zero bits is represented as -1 minus the sum of the weights of the zero-bits, a negative integer. For example, the infinite bit-vector #*01011... can be represented by the integer -6. Hence, logical operations on infinite bit-vectors with a finite number of one-bits or zero-bits can be performed by applying similar logical operations on their integer representations using the functions described below.

The above method of using integers to represent bit-vectors can also be used to represent sets. Suppose that set S is a subset of the universal set U. Then set S can be represented by a bit-vector in which each bit represents an element of U, and bit j is a one-bit if the corresponding element is also an element of S. In this way all finite subsets of U can be represented by positive integers. Similarly, all sets whose complements are finite can be represented by negative integers. The functions **logior**, **logand**, and **logxor** can then be used to compute the union, intersection and symmetric difference operations on sets represented in this way.

6.4.8.2 Functions Returning Result of Bit-wise Logical Operations

The logical operations performed by sixteen of the functions in this group can also be performed by the single function **boole**. This can be useful when it is necessary to parameterize a procedure so that it can use one of several logical operations.

logior &rest *integers*	Returns bit-wise logical *inclusive or* of its arguments*
logxor &rest *integers*	Returns the bit-wise logical *exclusive or* of its arguments*
logand &rest *integers*	Returns bit-wise logical *and* of its arguments*
logeqv &rest *integers*	Returns the bit-wise logical equivalence (*exclusive nor*) of its arguments*
lognand *integer1 integer2*	Returns the logical *not-and* of its arguments*
lognor *integer1 integer2*	Returns the logical *not-or* of its arguments*
logandc1 *integer1 integer2*	Returns the *and* complement of argument 1 with argument 2*
logandc2 *integer integer2*	Returns the *and* of argument 1 with the complement of argument2*
logorc1 *integer1 integer2*	Returns the *or* complement of *integer1* with *integer2**
logorc2 *integer1 integer2*	Returns the *or* of *integer1* with the complement of *integer2**
boole *op integer1* &rest *more-integers*	Generalization of other logical operations such as **logand, logior**, and **logxor**
lognot *integer*	Returns the logical complement of *integer*
ash *number count*	Shifts *number count* bits left or right depending on sign of *count*
zl:logand *number* &rest *more-numbers*	Returns bit-wise logical *and* of its arguments
zl:logior *number* &rest *more-numbers*	Returns bit-wise logical *inclusive or* of its arguments
zl:logxor *number* &rest *more-numbers*	Returns the bit-wise logical *exclusive or* of its arguments

*See the section "Comparison of Bit-wise Logical Operations", page 118.

6.4.8.3 Functions Returning Components or Characteristics of Argument

integer-length *integer*	Returns the number of significant bits in *integer*
logcount *integer*	Returns the number of one-bits in *integer*
zl:haipart *x n*	Depending on sign of *n*, returns the high or the low *n* bits of *x*
zl:haulong *x*	Returns the number of significant bits in *x* (similar to **integer-length**)

6.4.8.4 Predicates for Testing Bits in Integers

logbitp *index integer*	Returns true if *index* bit in *integer* (the bit whose weight is 2^{index}) is a one-bit
logtest *integer1 integer2*	Returns true if any 1-bits in *integer1* are 1-bits in *integer2*
zl:bit-test *x y*	Returns true if any 1 bits in *x* are 1 bits in *y* (same as **logtest**)

6.4.8.5 Comparison of Bit-wise Logical Operations

Argument1	0	0	1	1	
Argument2	0	1	0	1	*Operation Name*
logior	0	1	1	1	inclusive or
logxor	0	1	1	0	exclusive or
logand	0	0	0	1	and
logeqv	1	0	0	1	equivalence (exclusive nor)
lognand	1	1	1	0	nand (complement of and)
lognor	1	0	0	0	nor (complement of inclusive or)
logandc1	0	1	0	0	and complement of arg1 with arg2
logandc2	0	0	1	0	and arg1 with complement of arg2
logorc1	1	1	0	1	or complement of arg1 with arg2
logorc2	1	0	1	1	or arg1 with complement of arg2

6.4.9 Byte Manipulation Functions

Like logical operations, byte-manipulation functions are bit-wise operations that require integers as arguments. These functions operate on the internal binary representation of the integers, which are treated as though they were "sign-extended".

Byte manipulation functions deal with an arbitrary-width field of contiguous bits appearing anywhere in an integer. Such a contiguous set of bits is called a *byte.*

Note that we are not using the term *byte* to mean eight bits, but rather any number of bits within a number. These functions use *byte specifiers* to designate a specific byte position within any word. A byte specifier consists of the *size* (in bits) and *position* of the byte within the number, counting from the right in bits. A position of zero means that the byte is at the right end of the number. Byte specifiers are built using the **byte** function.

For example, the byte specifier **(byte 8 0)** refers to the lowest eight bits of a word, and the byte specifier **(byte 8 8)** refers to the next eight bits.

Bytes are extracted from and deposited into 2's complement signed integers. Treating the integers as signed means that negative numbers conceptually have infinitely many one-bits on the left. Bytes, being a finite number of bits, are never negative.

6.4.9.1 Summary of Byte Manipulation Functions

byte *size position*	Creates a byte specifier
byte-position *bytespec*	Extracts the position field of its argument
byte-size *bytespec*	Extracts the size field of its argument
dpb *newbyte bytespec integer*	Deposit byte; returns a copy of *integer* that is the same as *integer*, except in the bits specified by *bytespec*
deposit-field *newbyte bytespec integer*	Like **dpb**, except that *newbyte* is not taken to be right-justified
ldb-test *bytespec integer*	Returns true if the designated field is nonzero
ldb *bytespec integer*	Load byte; extracts byte of *integer* as specified by *bytespec*
mask-field *bytespec integer*	Similar to **ldb**, except for the position of the returned byte
deposit-byte *into-value position size byte-value*	Like **dpb**, except that byte specifier information is passed in separate arguments
load-byte *from-value position size*	Like **ldb**, except that byte specifier information is passed in separate arguments

6.4.10 Random Number Generation

The functions in this section provide a pseudorandom number generator facility. The basic function you use is **random** *n* &optional *state*. This function accepts a positive number *n* (integer or floating-point), and returns a pseudorandom number of the same type between zero (inclusive) and *n* (exclusive). The pseudorandom numbers generated are nearly uniformly distributed. If *n* is an integer, each of the possible results occurs with a probability very close to 1/*n*.

Between calls, the state of the pseudo-random-number generator is saved in a data structure of type **random-state**, stored in the variable ***random-state***. If you call **random** without supplying a value for *state*, **random** uses the value of ***random-state***.

Usually there is only one random-state, but there are functions that allow manipulation of this object to let you generate a reproducible sequence of random numbers within a program. ***random-state*** can be bound to any random-state object; it can also be printed out and successfully read back in.

Function **make-random-state** creates a new random-state data structure which can be used as the value of *state*. To copy the current random-number state object rather than create a new one, call **make-random-state** without an argument.

Use predicate **random-state-p** to test whether a given object is of type **random-state**.

To reproduce sequences of random numbers within a program you can create a random-state object and write it to a file with **print**; before running the program, **read** a copy of the random-state object from the printed representation in the file, then use this object to initialize the random-number generator for the program. Or, you can copy the random state directly via **make-random-state**.

Examples:

```
(random 2) => 0
(random 2) => 1
(random 3.0) => 1.1938573
(random 3.0) => 2.1395636
(random 1.0d0) => 0.5454759425745741d0

(setq base-random-state (make-random-state)) => #.(RANDOM-STATE...)
(setq copy1-base
      (make-random-state base-random-state)) => #.(RANDOM-STATE...)
(+ 1 (random 6 copy1-base)) => 3        ;simulate a roll of a die
(+ 1 (random 6 copy1-base)) => 6
(+ 1 (random 6 copy1-base)) => 4
(+ 1 (random 6 copy1-base)) => 2
```

```
(setq copy2-base
      (make-random-state base-random-state)) => #.(RANDOM-STATE...)
(+ 1 (random 6 copy2-base)) => 3        ;the same results
(+ 1 (random 6 copy2-base)) => 6
(+ 1 (random 6 copy2-base)) => 4
(+ 1 (random 6 copy2-base)) => 2
```

6.4.10.1 Random Numbers In Zetalisp

This section describes the pseudorandom number generator facility in Zetalisp. The function **zl:random** returns a new pseudorandom number each time it is called. Between calls, its state is saved in a data object called a *random-array.* Usually there is only one random-array; however, if you want to create a reproducible series of pseudorandom numbers, and be able to reset the state to control when the series starts over, then you need some of the other functions here.

A random-array consists of an array of numbers, and two pointers into the array. The length of the array is denoted by *length* and the distance between the pointers by *offset.* This algorithm produces well-distributed random numbers if *length* and *offset* are chosen carefully, so that the polynomial *x^length+x^offset+1* is irreducible over the mod-2 integers. The system uses 71 and 35.

The contents of the array of numbers should be initialized to anything moderately random, to make the algorithm work. The contents get initialized by a simple random number generator, based on a number called the *seed.* The initial value of the seed is set when the random-array is created, and it can be changed via function **si:random-initialize.** To have several different controllable resettable sources of random numbers, you can create your own random-arrays with function **si:random-create-array.** If you don't care about reproducibility of sequences, just use **zl:random** without the *random-array* argument.

6.4.10.2 Random Number Functions

make-random-state &optional *state*	Returns a new object of type **random-state**
random *number* &optional *state*	Returns a noncomplex number of the same kind as *number*
random-state-p *object*	Returns true if *object* is of type **random-state**
zl:random &optional *arg random-array*	Returns a random integer
si:random-create-array *length offset seed* &optional (*area* **nil**)	Creates, initializes, and returns an object of type *random-array*
si:random-initialize *array* &optional *new-seed*	Reinitializes the contents of *array* from *seed*

6.4.11 Machine-Dependent Arithmetic

Sometimes it is desirable to have a form of arithmetic that has no overflow checking (which would produce bignums), and truncates results to the word size of the machine. In Symbolics Common Lisp, this is provided by the following set of functions.

These functions should *not* be used for "efficiency"; they are probably less efficient than the Functions that *do* check for overflow. They are intended for algorithms which require this sort of arithmetic, such as hash functions and pseudorandom number generation.

6.4.11.1 Machine-dependent Arithmetic Functions

sys:%32-bit-plus *fixnum1 fixnum2*	Returns the sum of *fixnum1* and *fixnum2* in two's complement arithmetic
sys:%32-bit-difference *fixnum1 fixnum2*	Returns the difference of *fixnum1* and *fixnum2* in two's complement arithmetic
lsh *number count*	Requires fixnum arguments; returns *number* shifted *count* bits, left or right, depending on the sign of *count*; bits shifted at either end are lost.
rot *x y*	Returns *x* rotated *y* bits in a 32-bit field.
sys:%logdpb *newbyte bytespec integer*	Like **dpb**, except that it returns fixnums, thus reflecting changes in the sign bit.
sys:%logldb *bytespec integer*	Like **ldb**, except that it only loads out of fixnums and allows up to 32-bit byte size; thus the result can be negative.

7. Symbols and Keywords

7.1 Overview of Symbols

A *symbol* is a Lisp object in the Lisp environment. A symbol has a *print name*, a *value* (or *binding*), a *definition* (or the contents of its *function cell*), a *property list*, and a *package*. It is important to understand that a symbol can be any Lisp object, for example a variable, a function, or a list. It is also important to keep in mind that while we sometimes say that a symbol is the name of some object, a *name* is actually the printed representation of that object. A symbol is the object itself.

7.2 The Print Name of a Symbol

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the string is typed in to **read**, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, **print** displays the print-name.

7.2.1 How the Reader Recognizes Symbols

A string of letters, numbers, and "extended alphabetic" characters is recognized by the reader as a symbol, provided it cannot be interpreted as a number. Alphabetic case is ignored in symbols; lowercase letters are translated to uppercase. When the reader sees the printed representation of a symbol, it *interns* it on a *package*. See the section "Packages", page 635.

Symbols can start with digits; for example, **read** accepts one named "345T". If you want to put strange characters (such as lowercase letters, parentheses, or reader macro characters) inside the name of a symbol, put a slash before each strange character. If you want to have a symbol whose print-name looks like a number, put a slash before some character in the name. You can also enclose the name of a symbol in vertical bars, which quotes all characters inside, except vertical bars and slashes, which must be quoted with slash.

Examples of symbols:

```
foo
bar/(baz/)
34w23
|Frob Sale|
```

When a token could be read as either a symbol or an integer in a base larger than ten, the reader's action is determined by the value of **si:*read-extended-ibase-unsigned-number*** and **si:*read-extended-ibase-signed-number***.

7.2.2 Printed Representation of Symbols

If slashification is off, the printed representation of a symbol is simply the successive characters of the print-name of the symbol. If slashification is on, two changes must be made.

1. The symbol might require a package prefix for **read** to work correctly, assuming that the package into which **read** reads the symbol is the one in which it is being printed. (See the section "System Packages", page 663.)
2. If the printed representation would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), the printed representation must have one of the following kinds of quoting for those characters.
    - Slashes ("/") before each special character
    - Vertical bars ("|") around the whole name

The decision whether quoting is required is made using the readtable, so it is always accurate provided that ***readtable*** has the same value when the output is read back in as when it was printed. See the variable ***readtable*** in *Reference Guide to Streams, Files, and I/O.*

Uninterned symbols are printed preceded by #:. You can turn this off by evaluating **(setf (si:pttbl-uninterned-prefix *readtable*) "")**.

Functions Relating to the Print Name of a Symbol

symbol-name *sym*	Returns the print-name of *sym*.
zl:get-pname *sym*	Like **symbol-name**.
string= *sym1 sym2*	Checks to see if the print-names of *sym1* and *sym2* are the same.
zl:samepnamep *sym1 sym2*	Checks to see if the print-names of *sym1* and *sym2* are **string=**.

7.3 The Value Cell of a Symbol

Each symbol has associated with it a *value cell*, which refers to one Lisp object. This object is called the symbol's *binding* or *value*, since it is what you get when you evaluate the symbol. The binding of symbols to values allows symbols to be used as the implementation of *variables* in programs.

The value cell can also be *empty*, referring to *no* Lisp object, in which case the symbol is said to be *unbound.* This is the initial state of a symbol when it is created. An attempt to evaluate an unbound symbol causes an error.

Symbols are often used as special variables. See the section "Namespace System Variables" in *Networks.* The symbols **nil** and **t** are always bound to themselves; they cannot be assigned, bound, or otherwise used as variables. Attempting to change the value of **nil** or **t** (usually) causes an error.

The functions described here work on *symbols*, not *variables* in general.

Functions to assign a value to a symbol

set *symbol value*	Assign *value* to *symbol*
zl:set-globally *var value*	Assign *value* to *var* as a global variable.

Functions to retrieve the value of a symbol

symbol-value *sym*	Returns the current value of *sym.*
zl:symeval *sym*	Like
symbol-value-globally *var*	Returns the value of global variable *var* regardless of its current binding.
zl:symeval-globally *var*	Like **symbol-value-globally.**

Functions to remove the value of a symbol

makunbound *sym*	Remove the value from *sym.*
makunbound-globally *var*	Remove the value from global variable *var.*
variable-makunbound *variable*	Remove the value from *variable.*

Predicates for checking if a symbol has a variable

boundp *sym*
variable-boundp *variable*

Functions for locating the value cell of a symbol

zl:variable-location *variable*	Return a locative pointer to the value cell of *variable.*
zl:value-cell-location *sym*	Obsolete, use **zl:variable-location.**

7.4 The Function Cell of a Symbol

Every symbol also has associated with it a *function cell.* The *function* cell is similar to the *value* cell; it refers to a Lisp object. When a function is referred to by name, that is, when a symbol is *applied* or appears as the car of a form to be evaluated, that symbol's function cell is used to find its *definition*, the functional object that is to be applied. For example, when evaluating (+ **5 6**), the evaluator looks in +'s function cell to find the definition of +, in this case a compiled function, to apply to 5 and 6.

Like the value cell, a function cell can be empty, and it can be bound or assigned. (However, to bind a function cell you must use the **zl:bind** subprimitive.) The following functions are analogous to the similar value-cell-related functions. See the section "The Value Cell of a Symbol", page 124.

Functions Relating to the Function Cell of a Symbol

fboundp *sym*	Checks to see if *sym* is defined.
fmakunbound *sym*	Removes the definition from *sym*.
zl:fset *sym definition*	Stores *definition* in the function cell of *sym*.
symbol-function *sym*	Returns the function definition of *sym*.
zl:fsymeval *sym*	Like **symbol-function**.
sys:function-cell-location *sym*	Returns a locative pointer to *sym*'s function cell.

Since functions are the basic building block of Lisp programs, the system provides a variety of facilities for dealing with functions. See the section "Functions", page 251.

7.5 The Property List of a Symbol

Every symbol has an associated property list. See the section "Property Lists", page 139. When a symbol is created, its property list is initially empty.

The Lisp language itself does not use a symbol's property list for anything. (This was not true in older Lisp implementations, where the print-name, value-cell, and function-cell of a symbol were kept on its property list.) However, various system programs use the property list to associate information with the symbol. For instance, the editor uses the property list of a symbol that is the name of a function to remember where it has the source code for that function, and the compiler uses the property list of a symbol which is the name of a special form to remember how to compile that special form.

Because of the existence of print-name, value, function, and package cells, none of the Maclisp system property names (**zl:expr**, **zl:fexpr**, **macro**, **zl:array**, **subr**, **lsubr**, **fsubr**, and in former times **value** and **pname**) exist in Symbolics Common Lisp.

Functions Relating to the Property List of a Symbol

symbol-plist *sym*	Returns the property list of *sym*.
zl:plist *sym*	Like **symbol-plist**.
getf *plist indicator*	Searches for the property *indicator* on *plist*.
get-property *plist indicator-list*	Searches for a property (on *indicator-list*) on *plist*.

remprop *symbol indicator*	Removes *indicator* from the property list of *symbol.*
remf *plist indicator*	Removes the property *indicator* from *plist.*
zl:setplist *sym list*	Sets the property list of *sym* to *list.*
sys:property-cell-location *sym*	Returns a locative pointer to *sym*'s property list cell.

7.6 The Package Cell of a Symbol

Every symbol has a *package cell* that is used, for interned symbols, to point to the package to which the symbol belongs. For an uninterned symbol, the package cell contains **nil.**

7.6.1 Functions That Find the Home Package of a Symbol

symbol-package *symbol*	Return the package in which *symbol* resides
sys:package-cell-location *symbol*	Return a locative pointer to *symbol*'s package cell.
keywordp *object*	Check if *object* is a symbol in the keyword package.

7.7 Creating Symbols

The functions in this section are primitives for creating symbols. However, before discussing them, it is important to point out that most symbols are created by a higher-level mechanism, namely the reader and the **intern** function. Nearly all symbols in Lisp are created by virtue of the reader's having seen a sequence of input characters that looked like the printed representation of a symbol. When the reader sees such a printed representation, it calls **intern,** which looks up the sequence of characters in a big table and sees whether any symbol with this print-name already exists. If it does, **read** uses the existing symbol. If it does not exist, then **intern** creates a new symbol and puts it into the table, and **read** uses that new symbol.

A symbol that has been put into such a table is called an *interned* symbol. Interned symbols are normally created automatically; the first time someone (such as the reader) asks for a symbol with a given print-name, that symbol is automatically created.

These tables are called *packages.* In Symbolics Common Lisp, interned symbols are handled by the *package* system.

An *uninterned* symbol is a symbol used simply as a data object, with no special cataloging. An uninterned symbol prints the same as an interned symbol with the same print-name, but cannot be read back in.

The following functions can be used to create uninterned symbols explicitly.

Functions for Creating Symbols

make-symbol *pname*	Creates an uninterned symbol with print-name *pname*.
copy-symbol *sym copy-props*	Creates an uninterned symbol with the same print-name as *sym*.
zl:copysymbol *sym copy-props*	Like **copy-symbol**.
gensym	Invents a print-name and creates a symbol with that print-name.
zl:gensym	Like **gensym**.
gentemp	Like **gensym** but also interns the new symbol.

7.8 Introduction to Keywords

Keywords are disjoint from ordinary symbols. They are implemented as symbols whose home package is the **keyword** package, which has the empty string as a nickname. See the section "Package Names", page 645. Hence the printed representation of a keyword, a symbol preceded by a colon, is actually just a qualified name. As a matter of style, keywords are never imported into other packages and the **keyword** package is never inherited (used) by another package.

The only aspects of symbols significant to keywords are name and property list; otherwise, keywords could just as easily be some other data type. (Note that keywords are referred to as enumeration types in some other languages.)

The set of keywords is user-extensible; simply reading the printed representation of a new keyword is enough to create it. As a syntactic convenience, every keyword is a constant that evaluates to itself (just like numbers and strings). This eliminates the need to write a lot of " ' " marks when calling a function that takes **&key** arguments, but makes it impossible to have a variable whose name is a keyword. However, there is no desire to use keywords as names of variables (or of functions), because the colon would look ugly. In fact, no syntactic words of the Lisp language are keywords. Names of special forms, the **otherwise** that goes at the end of a **zl:selectq**, the **lambda** that identifies an interpreted function, names of declarations such as **special** and **arglist**, all are not keywords.

Using Keywords

Keywords can be used as symbolic names for elements of a finite set. For example, when opening a file with the **open** function you must specify a direction. The various directions are named with keywords, such as **:input** and **:output**.

One of the most common uses of keywords is to name arguments to functions that take a large number of optional arguments and therefore are inconvenient to call with arguments identified positionally. Each argument is preceded by a keyword that tells the function how to use that argument. When the function is called, it compares each keyword that was passed to it against each of the keywords it knows, using **eq**.

Another common use for keywords is as names for messages that are passed to active objects such as instances. When an instance receives a message, it compares its first argument against all the message names it knows, using **eq**. The practice of performing operations on flavor instances by sending messages to them has been superseded by generic functions. However, sending messages is still supported for compatibility. See the section "Using Message-Passing Instead of Generic Functions", page 471.

Since two distinct keywords cannot have the same name, keywords are not used for applications in which name conflicts can arise. For example, suppose a program stores data on the property lists of symbols. The data are internal to the program but the symbols can be global. An example of this would be a program-understanding program that puts some information about each Lisp function and special form on the symbol that names that function or special form. The indicator used should not be a keyword, because some other program might choose the same keyword to store its own internal data on the same symbol, causing a name conflict.

It is permissible, and in fact quite common, to use the same keyword for two different purposes when the two purposes are always separable by context. For instance, the use of keywords to name arguments to functions does not permit the possibility of a name conflict if you always know what function you are calling.

To see why keywords are used to name **&key** arguments, consider the function **make-array**, which takes one required argument followed by any number of keyword arguments. For example, the following specifies, after the first required argument, two options with names **:leader-length** and **:type** and values **10** and **sys:art-string**.

```
(make-array 100 :leader-length 10 :type 'art-string)
```

The file containing **make-array**'s definition is in the **system-internals** package, but the function is accessible to everyone without the use of a qualified name because the symbol **make-array** is itself inherited from **common-lisp-global**. But all the keyword names, such as **type**, are short and should not have to exist in

common-lisp-global where they would either cause name conflicts or use up all the "good" names by turning them into reserved words. However, if all callers of **make-array** had to specify the options using long-winded qualified names such as **system-internals:leader-length** and **system-internals:type** (or even **si:leader-length** and **si:type**) the point of making **make-array** global so that one can write **make-array** rather than **system-internals:make-array** would be lost. Furthermore, by rights one should not have to know about internal symbols of another package in order to use its documented external interface. By using keywords to name the arguments, we avoid this problem while not increasing the number of characters in the program, since we trade a "'" for a ":".

The data type names used with the **typep** function and the **typecase** and **zl:check-arg-type** special forms are sometimes keywords and sometimes not keywords. The names of data types that are built into the machine, such as **:symbol**, **:list**, **:fixnum**, and **:compiled-function**, are keywords. On the other hand, the names of data types that are defined as flavors or structures, such as ***package*** or **tv:window**, are not keywords. This unfortunate anomaly exists for historical reasons and is removed by Common Lisp, where names of data types, like names of functions, are never keywords.

When in doubt as to whether or not a symbol of the language is supposed to be a keyword, check to see whether it is documented with a colon at the front of its name.

8. Lists

8.1 Introduction to Lists

Lists are sequences represented in the form of linked cells called *conses*. There is a special object (the symbol **nil**) that is the empty list. All other lists are built recursively by adding a new element to the front of an existing list. This is done by creating a new *cons*, which is a record structure having two components called the *car* and the *cdr*. The *car* may hold anything, and the *cdr* may point to the previously existing list. A list, then is a chain of conses linked by their *cdr* components and terminated by **nil**.

The *car* components of the conses in a list are called the *elements* of the list. For each element of the list there is a cons. The empty list has no elements at all.

A *dotted list* is one whose cons does not have **nil** for its *cdr*, but has some other data object (which is also not a cons) as its *cdr*. Such a list is called "dotted" because of the special notation for it. See the section "Printed Representation of Conses", page 132.

Often the term *list* is used to refer either to true lists or to dotted lists. The term "true list" means a list terminated by **nil**. Most functions that operated on lists require their arguments to be true lists.

A *tree* is a cons and all other conses transitively accessible to that cons through *car* and *cdr* links, going down through the links until non-conses are reached at the end of the branches. The non-conses so reached are called the *leaves* of the tree.

8.2 How the Reader Recognizes Conses

When **read** sees an open parenthesis, it knows that the printed representation of a cons is coming, and calls itself recursively to get the elements of the cons or the list that follows. Any of the following are valid:

```
(foo . bar)
(foo bar baz)
(foo . (bar . (baz . nil)))
(foo bar . quux)
```

The first is a cons whose *car* and *cdr* are both symbols. The second is a list, and the third is exactly the same as the second (although **print** would never produce it). The fourth is a "dotted list"; the *cdr* of the last cons cell (the second one) is not **nil**, but **quux**.

Whenever the reader sees any of the above, it creates new cons cells; it never returns existing list structure. This contrasts with the case for symbols, as very often **read** returns symbols that it found interned in the package rather than creating new symbols itself. Symbols are the only thing that work this way.

The dot that separates the two elements of a dotted-pair printed representation for a cons is only recognized if it is surrounded by delimiters (typically spaces). Thus dot can be freely used within print-names of symbols and within numbers.

Tokens that consist of more than one dot, but no other characters, are valid symbols in Zetalisp but errors in Common Lisp. For Common Lisp, the variable **si:*read-multi-dot-tokens-as-symbols*** should be set to **nil**. See the variable **si:*read-multi-dot-tokens-as-symbols*** in *Symbolics Common Lisp: Language Dictionary*.

Zetalisp Note: If the circle-X (⊗) character is encountered, it is an octal escape, which might be useful for including unusual characters in the input. The next three characters are read and interpreted as an octal number, and the character whose code is that number replaces the circle-X and the digits in the input stream. This character is always taken to be an alphabetic character, just as if it had been preceded by a slash.

8.3 Printed Representation of Conses

The printed representation for conses tends to favor lists. It starts with an open-parenthesis. Then the *car* of the cons is printed and the *cdr* of the cons is examined. If the *cdr* is **nil**, a close-parenthesis is printed. If the *cdr* is anything else but a cons, "space dot space" followed by that object is printed. If the *cdr* is a cons, we print a space and start all over (from the point *after* we printed the open-parenthesis) using this new cons. Thus, a list is printed as an open-parenthesis, the printed representations of its elements separated by spaces, and a close-parenthesis.

This is how the usual printed representations such as **(a b (foo bar) c)** are produced.

The following additional feature is provided for the printed representation of conses: as a list is printed, the printing functions maintain the length of the list so far, and the depth of recursion of printing lists. If the length exceeds the value of the variable ***print-length***, **print** terminates the printed representation of the list with an ellipsis (three periods) and a close-parenthesis. If the depth of recursion exceeds the value of the variable ***print-level***, the portion of the list beyond the specified depth is printed as "#". These two features allow a kind of abbreviated printing that is more concise and suppresses detail. Of course, neither the ellipsis nor the "#" can be interpreted by **read**, since the relevant information is lost.

Note that all of the printing functions no longer use **zl:prinlevel** and **zl:prinlength** to control printing.

8.4 Type Specifiers and Type Hierarchy for Lists

The data types associated with lists are:

null cons list sequence

Here are descriptions of these Symbolics Common Lisp data types:

null	A primitive Lisp data type whose sole object is **nil**, the empty list.
cons	A primitive Lisp data type that consists of a *car* and a *cdr*. If the *car* and *cdr* of the cons are both **nil**, then the cons is the representation of the empty list, and can be reduced to the symbol **nil**.
list	A sequence of linked conses, which is built by recursively adding new conses to an existing list. A list can be recursively defined to be the symbol **nil**, or a cons whose *cdr* is a list. There is a special object (the symbol **nil**) that is the empty list. Note that *list*, which is not a *primitive* Lisp data type, is taken to mean the union of the *cons* and *null* data types; therefore, it encompasses both true lists and dotted lists.
sequence	A supertype of the **list** and **vector** (one-dimensional array) types. These types have the common property that they are ordered sets of elements. Functions that can be used on sequences can also be used on lists.

In summary: objects of the type *list* are a subset of objects of the type *sequence*, and the subsets of the type *list* are the types *cons* and *null*.

Here are descriptions of other objects that are related to lists, either being represented by them or being part of their structure:

alist	An *association list*, or *alist*, is a data structure consisting of a list of pairs of conses where each pair is an association. The *car* of the pair is called the *key*, and the *cdr* is called the *datum*. It is often represented as a list of dotted conses.
car	This is the first element of a cons. It can be a symbol, another cons, or the symbol **nil** (this is only the case when the cons

	represents the empty list). It is also the item returned when you use the **car** function on a list.
cdr	This is the second element of a cons. It is usually a pointer to the next cons in a list, or the symbol **nil**, representing the end of the list. However, it may also be another symbol, as in the case of a dotted list such as **(a . b)**. It is also the item returned when you use the **cdr** function on a list.
circular list	A *circular list* is like a list except that the *cdr* of the last cons, instead of being **nil**, is the first cons of the list. This means that the conses are all hooked together in a ring, with the *cdr* of each cons being the next cons in the ring. While these are perfectly good Lisp objects, and there are functions to deal with them, many other functions will have trouble with them. Functions that expect lists as their arguments often iterate down the chain of conses waiting to see a **nil**, and when handed a circular list this can cause them to compute forever. The ***print-circle*** variable is useful for printing circular lists.
dotted list	A *dotted list* is like the list data type, except that the last element of the list does not have to be **nil**. This name comes from the printed representation, which includes a "dot" character, such as **(a . b)**. The *car* of this dotted list is the symbol **a**, and the *cdr* of the list is the symbol **b**.
plist	A *property list*, or *plist*, is a tabular data structure consisting of a list of alternating keyword symbols (called *indicators*) and Lisp objects (called the *value* or *property*). There can be no duplication among indicators, since a plist can only have one property at a time with a given name. It is represented as an even-numbered list of elements.
set	*set* is a logical term that refers to a one-dimensional list. It refers to a list of symbols, as opposed to a list of lists. Therefore, the list **(a b c)** would be a set, but the list **(a (b c) d)** would not.
tree	A *tree* is any data structure made up of conses whose *car*s and *cdr*s are other conses. Another way of looking at a tree is as a list of lists.

Note that lists, dotted lists, trees, association lists, property lists, and circular lists are not mutually exclusive data types; they are different ways of looking at structures composed of conses.

Finally, for completeness, here is the description of the data type **atom**.

atom	Equivalent to the data type **symbol**. Every atom (or symbol) has a *print name*, a *value*, a *definition*, a *property list*, and a *package*. An atom can be a variable, a function, or any other Lisp object that is *not* a list. See the section "Symbols and Keywords", page 123.

Atoms are the leaves of trees.

8.5 Keyword Arguments That Delimit and Direct List Operations

Some list functions let you delimit the portion of the list to be operated on. Such functions have keyword arguments **:start** and **:end**, which must be non-negative integer indices into the list.

:start indicates the position for operation within the list. It defaults to zero (the first element in the list). If **:start** and **:end** are both present **:start** must be less than or equal to **:end** or an error is signalled.

:end indicates the position of the first element in the list *beyond* the end of the operation. It defaults to **nil** (the end of the list).

For search operations, one can specify the direction in which to search by using the keyword **:from-end**. Where **:from-end** is present, the function normally process the string in the forward direction, but if a non-**nil** value is specified for this keyword, processing is performed in the reverse direction.

In some functions, the keyword **:count** is used to specify how many occurrences of an item should be affected. If **:count** is **nil**, or not supplied, all matching items are affected.

Several functions used to create conses or lists use the keyword argument **:area**. The value of this keyword specifies in which area the object should be created. See the section "Areas" in *Internals, Processes, and Storage Management.* **:area** should be either an area number (an integer), or **nil** to mean the default area.

Some functions that create lists allow you to specify the items that are to be in the list. The keyword **:initial-element** (Common Lisp) or **:initial-value** (Zetalisp) effects this.

Most Common Lisp functions for searching through or otherwise operating on lists allow you to specify the kind of predicate to be used to identify a matching element. They also allow you to apply a function to an element before the predicate test. The keywords **:test**, **:test-not** and **:key** are used for these purposes.

:test specifies a binary operation to be applied to an argument supplied with the function and each of the elements of the target list in turn. If **:test** is not supplied the default operation is **eql**. For example,

```
    (adjoin item list)
```

returns a copy of *list* with *item* added to it if *list* did not already contain an element that was **eql** to *item.*

```
    (adjoin item list :test equal)
```

returns a copy of *list* with *item* added to it if *list* did not already contain an element that was **equal** to *item.*

:test-not reverses the sense of **:test**. For example,

```
    (adjoin item list :test-not equal)
```

returns a copy of *list* with *item* added to it if *list did* already contain an element that was **equal** to *item.*

If an operation tests elements of a list in any manner, the keyword argument **:key**, if not **nil**, should be a function of one argument that will extract from an element the part to be tested in place of the whole element. Note that operations that test elements included both those that use the **:test** and **:test-not** keywords and those that have -if and -if-not versions, for example, **nsubst-if** and **nsubst-if-not**.

In the following function descriptions, an element *target* of a list satisfies the test if:

- A basic function was called, *test-function* was specified by **:test**, *key-function* was specified by **:key**, and

  ```
  (funcall test-function target (key-function item))
  ```

  is true.

- A basic function was called, *test-function* was specified by **:test-not**, *key-function* was specified by **:key**, and

  ```
  (funcall test-function target (key-function item))
  ```

  is false.

- An -if function was called, and

  ```
  (funcall predicate
  (key-function item))
  ```

  is true.

- An -if-not function was called, and

  ```
  (funcall predicate
  (key-function item))
  ```

  is false.

8.6 Special Types of Lists

There are two types of lists that are special in the sense that Lisp contains a number of functions intended to operate on them. These are association lists and property lists.

8.6.1 Association Lists

An *association list,* or *alist,* is a structure made up of a list of pairs (or conses) in which each pair is an association. The car of each pair is the *key,* and the cdr is the *datum.*

There are a number of functions available for retrieving a datum, given the key.

```
((heron .  wader) (loon . diver) (eagle . raptor))
```

In this example, the association is between the bird name, and the class of bird. You can use the function **assoc** to get the class of bird, given the bird name. You can also use the function **rassoc** to do a reverse association and get the bird name given the class of bird.

Note that in this example we have a dotted list. It is a list of conses, each of which has a symbol for both its *car* and *cdr.*

An advantage of association lists is that they can be expanded simply by adding entries to the front. Because the searching function **assoc** searches in order from the front, new entries can shadow old ones.

In some cases, it is desirable to regard an association list as mapping in the reverse direction, that is, mapping *from* a datum *to* a key. The function *cl:rassoc* is useful for searching a list using this mapping.

8.6.1.1 Functions That Operate on Association Lists

The first two functions listed below are used to construct association lists. The remainder are used to extract a cons pair or list of pairs from an alist in accordance with some specified test.

acons *key datum alist*	Constructs a new association list by adding the pair (*key* . *datum*) onto the front of *alist.*
pairlis *keys data* &optional *a-list*	Takes two lists and makes an association list

	that associates elements of the first list to corresponding elements of the second list. If an optional *a-list* is supplied, the new pairs are added to the front of it.
zl:pairlis *var vals*	Takes two lists and makes an association list that associates elements of the first list to corresponding elements of the second list.
assoc *item a-list* &key (*test #'eql*) *test-not* (*key #'identity*)	Searches the association list *a-list* for *item.* Returns the cons whose *car* satisfies the test.
assoc-if *predicate a-list* &key *key*	Searches the association list *a-list* for the *car* of the pair satisfying *predicate.*
assoc-if-not *predicate a-list* &key *key*	Searches the association list *a-list* for *car* of the pair not satisfying *predicate.*
zl:assoc *item a-list*	Searches the association list *a-list* for *item* using **equal** as the comparing predicate. Returns the cons whose *car* satisfies the test.
zl:sassoc *item in-list function*	Like (**zl:assoc** *item alist*) except that if *item* is not found in *alist*, instead of returning **nil**, **zl:sassoc** calls the function *function* with no arguments.
zl:assq *item in-list*	Looks up *item* in the association list (list of conses) *alist* using **eq** as the comparing predicate. Returns the cons whose *car* satisfies the test.
zl:sassq *item in-list else*	Like (**zl:assq** *item alist*) except that if *item* is not found in *alist*, instead of returning **nil**, **zl:sassq** calls the function *function* with no arguments.
zl:ass *pred item list*	Same as **zl:assq** except that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison instead of **eq**.
zl:memass *pred item list*	Searches *alist* just like **zl:ass**, but returns the portion of the list beginning with the pair containing *item*, rather than the pair itself.

rassoc *item a-list* &key (*test* #'*eql*) *test-not* (*key* #'*identity*)
Reverse form of **assoc**. However, **rassoc** searches for the first pair whose *cdr*, rather than *car*, satisfies the predicate specified by the **:test** keyword.

zl:rassoc *item a-list*
Reverse form of **zl:assoc**. However, **rassoc** searches for the first pair whose *cdr*, rather than *car*, is **eq** to *item*.

rassoc-if *predicate a-list* &key *key*
Searches the association list *a-list* for the *cdr* of the pair satisfying *predicate*.

rassoc-if-not *predicate a-list* &key *key*
Searches the association list *a-list* for the *cdr* of the pair not satisfying *predicate*.

zl:rassq *item in-list*
Means "reverse assq." It is like zl:assq, but it tries to find an element of *alist* whose *cdr* (not *car*) is *eq* to *item*.

zl:rass *pred item in-list*
rass is to **zl:rassq** as **ass** is to zl:assq. That is, it takes a predicate to be used instead of **eq**.

8.6.2 Property Lists

Lisp has always had a kind of tabular data structure called a *property list* (plist for short). A property list contains zero or more entries; each entry associates from a keyword symbol (called the *indicator*) to a Lisp object (called the *value* or, sometimes, the *property*). There are no duplications among the indicators; a property list can only have one property at a time with a given name.

Property lists are very similar to association lists. (See the section "Association Lists", page 137.) The difference is that a property list is an object with a unique identity; the operations for adding and removing property list entries are side-effecting operations that alter the property list rather than making a new one. An association list with no entries would be the empty list (), that is, the symbol nil. There is only one empty list, so all empty association lists are the same object. Each empty property list is a separate and distinct object.

The implementation of a property list is a memory cell containing a list with an even number (possibly zero) of elements. Each pair of elements constitutes a *property*; the first of the pair is the indicator and the second is the value. The memory cell is there to give the property list a unique identity and to provide for side-effecting operations.

The term "property list" is sometimes incorrectly used to refer to the list of entries inside the property list, rather than the property list itself. We try to avoid such confusions wherever possible.

How do we deal with "memory cells" in Lisp; that is, what kind of Lisp object is a property list? Rather than being a distinct primitive data type, a property list can exist in one of three forms:

1. A property list can be a cons whose *cdr* is the list of entries and whose *car* is not used and is therefore available to the user to store something.

2. The system associates a property list with every symbol. See the section "The Property List of a Symbol", page 126. A symbol can be used where a property list is expected; the property-list primitives automatically find the symbol's property list and use it.

3. A property list can be a memory cell in the middle of some data structure, such as a list, an array, an instance, or a defstruct. An arbitrary memory cell of this kind is named by a locative. See the section "Cells and Locatives", page 29. Such locatives are typically created with the **locf** special form. See the macro **locf** in *Symbolics Common Lisp: Language Dictionary.*

Property lists of the first kind are called "disembodied" property lists because they are not associated with a symbol or other data structure. The way to create a disembodied property list is **(ncons nil)**, or **(ncons** *data***)** to store *data* in the car of the property list.

Here is an example of the list of entries inside the property list of a symbol named **b1** that is being used by a program that deals with blocks:

```
(color blue on b6 associated-with (b2 b3 b4))
```

There are three properties, and so the list has six elements. The first property's indicator is the symbol **color**, and its value is the symbol **blue**. We say that "the value of **b1**'s **color** property is **blue**", or, informally, that "**b1**'s **color** property is **blue.**" The program is probably representing the information that the block represented by **b1** is painted blue. Similarly, it is probably representing in the rest of the property list that block **b1** is on top of block **b6**, and that **b1** is associated with blocks **b2**, **b3**, and **b4**.

There is a mixin flavor, called **sys:property-list-mixin**, that provides messages that do things analogous to what the property list functions do.

8.6.2.1 Functions That Operate on Property Lists

The following functions add to, modify, or search property lists.

zl:putprop *sym value indicator*	Gives *sym* an *indicator*-property of *x*.
defprop *sym value indicator*	Form of **putprop** with "unevaluated arguments", which is sometimes more convenient for typing.
get *symbol indicator* &optional (*default* **nil**)	Searches the property list of *symbol* for an indicator that is **eq** to *indicator*. If the search fails, *default* is returned.
zl:get *symbol indicator*	Looks up *indicator* on the *symbol*'s property list. If it finds such a property, it returns the value; otherwise it returns **nil**.
zl:getl *symbol indicator-list*	Searches down *symbol* for any of the indicators in *indicator-list* until it finds a property whose indicator is one of the elements of *indicator-list*.
remprop *symbol indicator*	Removes *symbol*'s *indicator* property, by splicing it out of the property list.
zl:remprop *symbol indicator*	Removes *symbol*'s *indicator* property, by splicing it out of the property list. It returns that portion of the list inside *symbol* of which the form *indicator*-property was the *car*.

8.7 Operations with Lists

There are many types of list operations. Most of these can be done with specialized list functions, while some can be done with more general-purpose sequence functions. The majority of the list functions require true lists as arguments.

The list operations fall logically into nine major groups, as follows:

- Operating on Lists with Predicates
- Finding Information about Lists and Conses
- Constructing Lists and Conses
- Copying Lists
- Extracting from Lists

- Modifying Lists
- Comparing Lists
- Searching Lists
- Sorting Lists

8.7.1 Predicates That Operate on Lists

There are two groups of predicate functions for use with lists. In the first group are those that test the data type of their arguments. The first five entries in the following list are in this group. The remaining predicates test members of lists for some quality, except for **tree-equal**, which is used for comparisons.

atom *object*	Returns **t** if its argument is not a cons, otherwise **nil**.
consp *object*	Returns **t** if its argument is a cons, and otherwise is **nil**.
listp *object*	Returns **t** if its argument is a cons or the empty list (), otherwise **nil**.
zl:listp *object*	Returns **t** if its argument is a cons or *not* the empty list (), otherwise **nil**. Note **listp** and **zl:listp** are not equivalent.
nlistp *x*	Returns **t** if its argument is not a cons, otherwise **nil**. Identical to **atom**.
endp *object*	**endp** is the recommended way to test for the end of a list. Returns **nil** when it is applied to a cons, and **t** when it is applied to **nil**.
tailp *tail list*	Returns **t** if *tail* is an ending sublist of *list*, otherwise **nil**.
subsetp *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)	Checks if list1 is a subset of list2. With default test **eql** **subsetp** returns **t** if every element of *list1* appears in *list2*, otherwise **nil**.
tree-equal *x y* &key *test test-not*	Compares two trees of conses *x* and *y*. With default test **eql** returns **t** if *x* and *y* are isomorphic trees with identical leaves, otherwise **nil**.

some *predicate* &rest *lists*	Each element in each of *lists* is tested against *predicate*. Returns whatever value *predicate* returns as non-**nil**, as soon as any invocation of *predicate* returns a non-**nil** value. Otherwise returns **nil**.
zl:some *list predicate*	Each element in each of *list* is tested against *predicate*. Returns the tail of the list at the point at which value *predicate* returns as non-**nil**, Otherwise returns **nil**.
every *predicate* &rest *sequences*	Each element in *sequence* is tested against *predicate*. Returns **nil** as soon as any invocation of *predicate* returns **nil**, otherwise non-**nil**.
zl:every *list pred* &optional (*step #'cdr*)	Each element in *list* is tested against the *predicate*. Extraction from the list can be changed by the *step* function. Returns *t* if predicate returns non-**nil** when applied to every element of list, otherwise **nil** if predicate returns **nil** for some element.

8.7.2 Functions for Finding Information About Lists and Conses

These functions return the length of a list, the position of an item in a list, or the location of a cons's *car*.

length *sequence*	Counts the number of elements in *sequence*. Returns a non-negative integer.
zl:length *x*	Counts the number of elements in *x*. Returns a non-negative integer. Use equivalent **length**.
list-length *list*	Counts the number of elements in *list*. Returns a non-negative integer. Unlike **length** if *list* is circular list-length will return **nil**.
zl:find-position-in-list *item list*	Looks down *list* for an element that is **eq** to *item*, like **zl:memq**; however, it returns the numeric index in the list at which it found the first occurrence of *item*, or **nil** if it did not find it at all.

zl:find-position-in-list-equal *item list*	same as **zl:find-position-in-list**, except that the comparison is done with **equal** instead of **eq**.
zl:car-location *cons*	Returns a locative pointer to the cell containing the *car* of *cons*.

8.7.3 Functions for Constructing Lists and Conses

This group includes functions that construct conses and lists from scratch, as well as functions that make new lists by adding to or combining existing lists.

The functions that create conses, **cons**, **ncons**, and **xcons**, and their in-area variants can be used to construct normal, that is, not cdr-coded lists. The higher-level functions **list**, **make-list**, **append**, and their variants construct cdr-coded lists.

cons *x y*	Adds a new element to the front of a list. It is the primitive function to create a new *cons* whose *car* is *x* and whose *cdr* is *y*.
ncons *x*	Same as **(cons** *x* **nil)**.
xcons *y x*	Like **cons** except that the order of the arguments is reversed ("exchanged cons").
cons-in-area *x y area*	Creates a cons in a specific area.
ncons-in-area *x area*	Same as **(cons-in-area** *x* **nil** *area-number***)**.
xcons-in-area *y x area*	Same as **(cons-in-area** *y x area-number***)**
list &rest *elements*	Constructs and returns a list of its arguments.
list* &rest *args*	Same as **list** except that the last cons of the constructed list is "dotted." It must be given at least one argument.
list*-in-area *area* &rest *args*	Same as **list*** except that it takes an extra argument, an area number, and creates a list in that area.
list-in-area *area* &rest *elements*	Same as **list** except that it takes a extra argument, an area number, and creates the list in that area.
make-list *size* &key *initial-element area*	Creates and returns a list containing *size* elements.

zl:make-list *length* &key *area initial-value*	Creates and returns a list containing *length* elements. Use **make-list**.
circular-list &rest *args*	Constructs a circular list whose elements are *args*, repeated infinitely. Often used with mapping.
nconc &rest *arg*	Takes lists as arguments. Destructively concatenates and returns *args* in a list. See the function **concatenate** in *Symbolics Common Lisp: Language Dictionary.*
nreconc *l tail*	Same as **(nconc (nreverse** *l***)** *tail***)**.
append &rest *lists*	Concatenates the argument *lists* returning a new list.
revappend *x y*	Same as **(append (reverse** *x***)** *y***)**.
adjoin *item list* &key (*test #'eql*) *test-not* (*key #'identity*)	Adds an *item* to *list* provided that it is not already on the list. Returns a new list.
push *item reference* &key *area localize*	With the list held in *reference* viewed as a push-down stack **push** pushes an element onto the top of the stack.
zl:push *item list*	Adds and item to the front of a list that is stored in a generalized variable. Use **push** instead.
zl:push-in-area *item list area*	Adds an item to the front of a list that is stored in a generalized variable.
pushnew *item reference* &key *test test-not key*	With the list held in *reference* viewed as a push-down stack **pushnew** pushes *item* onto the top of the stack, unless it is already a member of the list.

8.7.4 Functions for Copying Lists

This group includes functions that copy conses, lists, or trees, including some system functions that help improve locality of reference. **copy-list** can be used to copy a list, converting it into compact cdr-coded form.

copy-list *list* &optional *area force-dotted*

This function returns a list that is **equal** to *list*, but not **eq**. (only the top level of list structure is copied).

zl:copylist *list* &optional *area force-dotted*

Returns a list that is **equal** to *list*, but not **eq**. **zl:copylist** does not copy any elements of the list: only the conses of the list itself.

copy-list* *list* &optional *area*

This function is the same as **copy-list** except that the last cons of the resulting list is never cdr-coded.

zl:copylist* *list* &optional *area*

This is the same as **zl:copylist** except that the last cons of the resulting list is never cdr-coded.

copy-alist *al* &optional *area*

This function returns an association list that is **equal** to *al*, but not **eq** (only the top level of list structure is copied).

zl:copyalist *al* &optional *area*

Copies an association list.

copy-tree *tree* &optional *area*

Copies a tree of conses.

zl:copytree *tree* &optional *area*

Copies all the conses of a tree and makes a new tree with the same fringe.

zl:copytree-share *tree* &optional *area* (*hash* (*make-equal-hash-table*)) *cdr-code*

zl:copytree-share is similar to **zl:copytree**. However, it also assures that all lists or tails of lists are optimally shared when **equal**.

sys:copy-if-necessary *thing* &optional *(default-cons-area working-storage-area)*

Moves *thing* from a temporary storage area or stack list to a permanent area (*thing* can be a list).

sys:localize-list *list* &optional *area*

A function that improves locality of incrementally-constructed lists and alists.

sys:localize-tree *tree* &optional (*n-levels 100*) *area*

A function that improves locality of incrementally-constructed lists and trees.

8.7.5 Functions for Extracting From Lists

This group includes functions that return a specified item or items from a list. The item is specified by its position in the list.

car *x*	Returns the first element of *x*, called the *car*.
cdr *x*	Returns the rest of the list after the first element, called the *cdr*.
c{a,d}*r *x*	An abbreviation for sequences of *car*s and *cdr*s.
first *list*	Returns the first element of the list. **first** is identical to **car**.
zl:firstn *n list*	**zl:firstn** returns a list of length *n*, whose elements are the first *n* elements of **list**.
second *list*	Returns the second element of the *list*.
third *list*	Returns the third element of the *list*.
fourth *list*	Returns the fourth element of the list.
fifth *list*	Returns the fifth element of the list.
sixth *list*	Returns the sixth element of the *list*.
seventh *list*	Returns the seventh element of the *list*.
eighth *list*	Returns the eighth element of the *list*.
ninth *list*	Returns the ninth element of the *list*.
tenth *list*	Returns the tenth element of the *list*.
last *list*	Returns the last cons of *list*.
nleft *n list* &optional *tail*	Returns the result of taking the **cdr** of *list* enough times so that taking *n* more cdrs would yield *tail*. When *tail* is **nil**, **zl:nlist** simply returns the last *n* elements of *list*.
nth *n object*	Returns the *n*th element of *list*, where the zeroth element is the car of the list.
nthcdr *n list*	Performs the **cdr** operation on the *list* *n* times, and returns the result.
rest *x*	Equivalent to **cdr**, but it mnemonically complements the function **first**.

zl:rest1 *list*	Returns the rest of the elements of a list, starting with element 1 (counting the first element as the zeroth).
zl:rest2 *list*	Returns the rest of the elements of a list, starting with element 2 (counting the first element as the zeroth).
zl:rest3 *list*	Returns the rest of the elements of a list, starting with element 3 (counting the first element as the zeroth).
zl:rest4 *list*	Returns the rest of the elements of a list, starting with element 4 (counting the first element as the zeroth).
some *predicate* &rest *lists*	Each element in each of *lists* is tested against *predicate*. Returns whatever value *predicate* returns as non-**nil**, as soon as any invocation of *predicate* returns a non-**nil** value. Otherwise returns **nil**.
pop *list*	The result returned by **pop** is the *car* of the contents of *list*, and as a side effect, the *cdr* of contents is stored back into *list*.

8.7.6 Functions for Modifying Lists

This group contains functions that that either modify list structures or return modified copies of a list structure. Those functions that change the original structure rather than make copies are referred to as "destructive." Their names begin with the letter *n* except for **delete**, which can be considered a destructive version of **remove**.

rplaca *cons x*	Changes the *car* of *x* to *y* and returns (the modified) *x*.
rplacd *cons x*	Changes the *cdr* of *x* to *y* and returns (the modified) *x*.
pop *list*	The result returned by **pop** is the *car* of the contents of *list*, and as a side effect, the *cdr* of contents is stored back into *list*.
zl:pop *list* &optional *dest*	Removes an element from the front of a list that is stored in a generalized variable.

butlast *x* &optional *(n 1)* — Creates and returns a list with the same elements as *list*, excepting the last element.

nbutlast *list* &optional *(n 1)* — Destructive version of **butlast**

remove *item sequence* &key (*test #'eql*) *test-not* (*key #'identity*) *from-end* (*start* 0) *end count*

Non-destructively removes items matching *item* from the *sequence*. Returns the new sequence.

zl:remove *item sequence* &optional (*ntimes most-positive-fixnum*)

Non-destructive version of **zl:delete**. Use **remove** instead.

zl:rem *predicate item list* &optional (*ntimes most-positive-fixnum*)

Non-destructively removes occurrences of *item* from *list* that satisfy the *predicate.*

delete *item list* &key (*test #'eql*) *test-not* (*key #'identity*) *from-end* (*start 0*) *end count*

Destructively removes items matching *item* from the *list.* Returns the modified sequence.

zl:delete *item list* &optional (*ntimes most-positive-fixnum*)

Deletes the first *ntimes* occurrences of *item* in the *list* (**equal** is used for the comparison). Returns the *list* with all occurrences of *item* removed. Use **delete** instead.

zl:rem-if *pred list* &rest *extra-lists* — Means "remove if this condition is true." See **zl:subset-not**.

zl:del-if *pred list* — Just like **zl:rem-if** except that it modifies *list* rather than creating a new list and it does not take an *extra-lists* **&rest** argument.

zl:rem-if-not *pred list* &rest *extra-lists*

Means "remove if this condition is not true." See **subset**.

zl:del-if-not *pred list* — Just like **zl:rem-if-not** except that it modifies *list* rather than creating a new list and it does not take an *extra-lists* **&rest** argument.

zl:del *pred item list* &optional (*ntimes -1*)

Returns the *list* with all occurrences of *item* removed. *pred* is used for the comparison (*pred* should take two arguments).

zl:delq *item list* &optional (*ntimes -1*)	Returns the *list* with all occurrences of *item* removed. **equal** is used for the comparison.
zl:remq *item list* &optional (*times most-positive-fixnum*)	Similar to **zl:delq**, except that the list is not altered; rather, a new list is returned.
zl:subset *pred list* &rest *extra-lists*	Means "remove if this condition is not true." **subset** refers to the function's action if *list* is considered to represent a mathematical set. See **zl:rem-if-not**.
zl:subset-not *pred list* &rest *extra-lists*	Means "remove if this condition is not true." **user::subset-not** refers to the function's action if *list* is considered to represent a mathematical set. See **zl:rem-if**.
sublis *alist tree* &rest *args* &key (*test #'eql*) *test-not* (*key #'identity*)	Makes non-destructive substitutions for objects in a tree (a structure of conses).
nsublis *alist tree* &rest *args* &key (*test #'eql*) *test-not* (*key #'identity*)	Destructive version of **sublis**.
zl:sublis *alist form*	Makes non-destructive substitutions for symbols in a tree (a structure of conses). The first argument to **zl:sublis** is an association list.
zl:nsublis *alist form*	Destructive version of **sublis**.
subst *new old tree* &rest *args* &key (*test #'eql*) *test-not* (*key #'identity*)	Makes a copy of *tree*, substituting *new* for every subtree or leaf of tree such that *old* and the subtree or leaf satisfy the predicate specified by the **:test** keyword.
zl:subst *new old tree*	Substitutes *new* for all occurrences of *old* in *tree*, and returns the modified copy of *tree*.
subst-if *new predicate tree* &rest *args* &key *key*	Makes a copy of *tree*, substituting *new* for every subtree or leaf of *tree* such that *old* and the subtree or leaf satisfy the test specified by *predicate*.

subst-if-not *new predicate tree* &rest *args* &key *key*
Makes a copy of *tree*, substituting *new* for every subtree or leaf of *tree* such that *old* and the subtree or leaf do not satisfy the test specified by *predicate*.

nsubst *new old tree* &rest *args* &key (*test #'eql*) *test-not* (*key #'identity*)
destructive version of **subst**.

zl:nsubst *new old tree* — destructive version of **zl:subst**.

nsubst-if *new predicate tree* &rest *args* &key *key*
Destructive version of **subst-if**.

nsubst-if-not *new predicate tree* &rest *args* &key *key*
Destructive version of **subst-if-not**.

reverse *sequence* — Reverses the elements of the *sequence*. Returns a new reversed sequence.

zl:reverse *list* — Reverses the elements of the *list*. Returns a new reversed list.

nreverse *sequence* — Destructive version of **reverse**. Returns a modified sequence.

zl:nreverse *list* — Destructive version of **list**. Returns a modified list.

8.7.7 Functions for Comparing Lists

This group contains functions that compare the elements of list structures and return lists of those elements that are similar or of those elements that are different according to specified tests. Note that the predicate function **tree-equal** can also be used to compare lists. All but the last of these functions perform set operations on lists.

union *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Takes two lists and returns a new list containing everything that is an element of either of the lists.

zl:union *list1 list2* — Takes two lists and returns a new list containing everything that is an element of either of the lists using **eq** for its comparisons.

nunion *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Destructive version of **union**.

zl:nunion *list1 list2* — Destructive version of **zl:union.**

intersection *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Takes two lists and returns a new list containing everything that is an element of both lists, as checked by the **:test** and **:test-not** keywords.

zl:intersection *list1 list2* — Takes two lists and returns a new list containing everything that is an element of both lists, using **eq** for comparison.

nintersection *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Destructive version of **intersection.**

zl:nintersection *list1 list2* — Destructive version of **zl:intersection.**

set-difference *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Non-destructive function which returns a list of elements of *list1* that do not appear in *list2.*

nset-difference *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Destructive version of **set-difference.**

set-exclusive-or *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Non-destructive function which returns a list of elements that appear in exactly one of *list1* and *list2.*

nset-exclusive-or *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
destructive version of **set-exclusive-or.**

ldiff *list sublist* — **ldiff** (meaning "list difference") returns a new list, whose elements are those elements of *list* that appear before *sublist.*

8.7.8 Functions for Searching Lists

Functions in this group search for a specified item within a list.

member *item list* &key (*test #'eql*) *test-not* (*key #'identity*)
Searches *list* for an element that satisfies the predicate specified by the **:test** keyword.

zl:member *item list* — Searches *list* for an element, using **equal** for the comparison.

member-if *predicate list* &key *key* — Searches for an element in *list* which satisfies *predicate.*

member-if-not *predicate list* &key *key*
Searches for the first element in *list* which does not satisfy *predicate.*

zl:memq *item in-list*	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*
zl:mem *pred item list*	Same as **zl:memq** except that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison instead of **eq**.

8.7.9 Functions for Sorting Lists

Several functions are provided for sorting arrays and lists. These functions use algorithms that always terminate no matter what sorting predicate is used, provided only that the predicate always terminates. The main sorting functions are not *stable*; that is, equal items might not stay in their original order. If you want a stable sort, use the stable versions. But if you do not care about stability, do not use them, since stable algorithms are significantly slower.

After sorting, the argument (either list or array) has been rearranged internally to be completely ordered. In the case of an array argument, this is accomplished by permuting the elements of the array, while in the list case, the list is reordered by **rplacds** in the same manner as **cl:nreverse**. Thus, if the argument should not be clobbered, you must sort a copy of the argument, obtainable by **zl:fillarray** or **cl:copy-list**, as appropriate. Furthermore, **cl:sort** of a list is like **zl:delq** in that it should not be used for effect; the result is conceptually the same as the argument but in fact is a different Lisp object.

Should the comparison predicate cause an error, such as a wrong type argument error, the state of the list or array being sorted is undefined. However, if the error is corrected the sort proceeds correctly.

The sorting package is smart about compact lists; it sorts compact sublists as if they were arrays. See the section "Cdr-Coding", page 154. An explanation of compact lists is in that section.

sort *sequence predicate* &key *key*	Destructively modifies *sequence* by sorting it according to an order determined by *predicate.*
zl:sort *sequence predicate*	Destructively modifies *sequence* by sorting it according to an order determined by *predicate.*

stable-sort *sequence predicate* &key *key*
Same as **sort**, however **stable-sort** guarantees

that elements considered equal by *predicate* will remain in their original order.

zl:stable-sort *sequence predicate* Same as **zl:sort**, however **stable-sort** guarantees that elements considered equal by *predicate* will remain in their original order.

zl:sortcar *x sort-lessp-predicate-on-car*

zl:sortcar is the same as **zl:sort** except that the predicate is applied to the *car*s of the elements of *x*, instead of directly to the elements of *x*.

zl:stable-sortcar *x sort-lessp-predicate-on-car*

Like **zl:sortcar**, but if two elements of *x* are equal, then those two elements remain in their original order.

8.8 Cdr-Coding

This section explains the internal data format used to store conses inside the Symbolics Lisp Machine. It is only important to read this section if you require extra storage efficiency in your program.

The usual and obvious internal representation of conses in any implementation of Lisp is as a pair of pointers, contiguous in memory. If we call the amount of storage that it takes to store a Lisp pointer a "word," then conses normally occupy two words. One word (say it is the first) holds the car, and the other word (say it is the second) holds the cdr. To get the *car* or *cdr* of a list, you just reference this memory location, and to change the *car* or cdr, you just 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 to **nil**. To optimize this particular case of using conses, the Symbolics Lisp Machine 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.

The way it works is that there is an extra two-bit field in every word of memory, called the "cdr-code" field. There are three meaningful values that this field can have, which are called cdr-normal, cdr-next, and cdr-nil. The regular, noncompact way to store a cons is by two contiguous words, the first of which holds the *car* and the second of which holds the cdr. In this case, the cdr code of the first word is cdr-normal. (The cdr code of the second word does not matter; 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 the last location whose cdr code is cdr-nil. The list is represented as a pointer to the first of the *n* words.

Now, how are the basic operations on conses defined to work based on this data structure? Finding the *car* is easy: you just 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 you find there. If the code is cdr-normal, then you add one 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 one 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**.

If you examine these rules, you find that they work fine even if you mix the two kinds of storage representation within the same list. There is no problem with doing that.

How about changing the structure? Like **car**, **rplaca** is very easy; you just store into the location addressed by the pointer. To do a **rplacd** you must read the location addressed by the pointer and examine the cdr code. If the code is cdr-normal, you just store into the location one greater than that addressed by the pointer; that is, you store into the second word of the two words. But if the cdr-code is cdr-next or cdr-nil, there is a problem: there is no memory cell that is storing the *cdr* of the cons. That is the cell that has been optimized out; it just does not exist.

This problem is dealt with by the use of "invisible pointers". An invisible pointer is a special kind of pointer, recognized by its data type (Symbolics Lisp Machine pointers include a data type field as well as an address field). The way they work is that when the Symbolics Lisp Machine reads a word from memory, if that word is an invisible pointer then it proceeds to read the word pointed to by the invisible pointer and use that word instead of the invisible pointer itself. Similarly, when it writes to a location, it first reads the location, and if it contains an invisible pointer then it writes to the location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually there are several kinds of invisible pointer that are interpreted in different ways at different times, used for things other than the cdr coding scheme.)

Here is how to do a **rplacd** when the cdr code is cdr-next or cdr-nil. Call the location addressed by the first argument to **rplacd** *l*. First, you allocate two contiguous words (in the same area that *l* points to). Then you store the old contents of *l* (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 *l*. (It does not matter what the cdr-code of this word is, since the invisible pointer data type is checked first.)

Now, whenever any operation is done to the cons (**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 address. So the newly allocated two-word cons is used for any operation done on the original object.

Why is any of this important to users? In fact, it is all invisible to you; everything works the same way whether or not compact representation is used, from the point of view of the semantics of the language. That is, the only difference that any of this makes is in efficiency. The compact representation is more efficient in most cases. However, if the conses are going to get **rplacd**'ed, then invisible pointers are created, extra memory is allocated, and the compact representation is seen to degrade storage efficiency rather than improve it. Also, accesses that go through invisible pointers are somewhat slower, since more memory references are needed. So if you care a lot about storage efficiency, you should be careful about which lists get stored in which representations.

You should try to use the normal representation for those data structures that are subject to **rplacd** operations, including **nconc** and **nreverse**, and the compact representation for other structures. The functions **cons**, **xcons**, **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 might get changed. Some functions, such as **sort**, take special care to operate efficiently on compact lists (**sort** effectively treats them as arrays). **nreverse** is rather slow on compact lists, since it simply uses **rplacd**.

(copy-list *list***)** is a suitable way to copy a list, converting it into compact form. See the function **copy-list** in *Symbolics Common Lisp: Language Dictionary*.

9. Arrays

The basic concepts and terminology associated with arrays are described elsewhere: See the section "Overview of Arrays", page 20.

In brief, an *array* is a Lisp object that consists of a group of elements, each of which is a Lisp object. *General arrays* allow the elements to be any type of Lisp object. *Specialized arrays* place constraints on the type of Lisp objects allowed as array elements.

The basic array functions enable you to create arrays (**make-array**), access elements (**aref**), alter elements (**setf** used with **aref**).

The individual elements of an array are identified by numerical *subscripts*. When accessing an element for reading or writing, you use the subscripts that identify that element. The number of subscripts used to refer to one of the elements of the array is the same as the dimensionality of the array. Thus, in a two-dimensional array, two subscripts are used to refer to an element of the array. The lowest value for any subscript is 0; the highest value depends on the array.

The number of dimensions of an array is called its *dimensionality*, or its *rank*. The dimensionality can be any integer from zero to seven, inclusive.

9.1 Type Specifiers and Type Hierarchy for Arrays

The type specifiers related to arrays include:

array	All arrays are of type **array**.
simple-array	An array that is not displaced, has no fill pointer, and is not adjustable after creation.
simple-string	A simple array whose elements are of type **character** or **string-char**.
vector	A one-dimensional array.
bit-vector	A vector whose elements are bits.
simple-vector	A vector that is not displaced, has no fill pointer, and is not adjustable after creation.
simple-bit-vector	A simple vector whose elements are bits.

Figure 3 shows the relationships among the various array types.

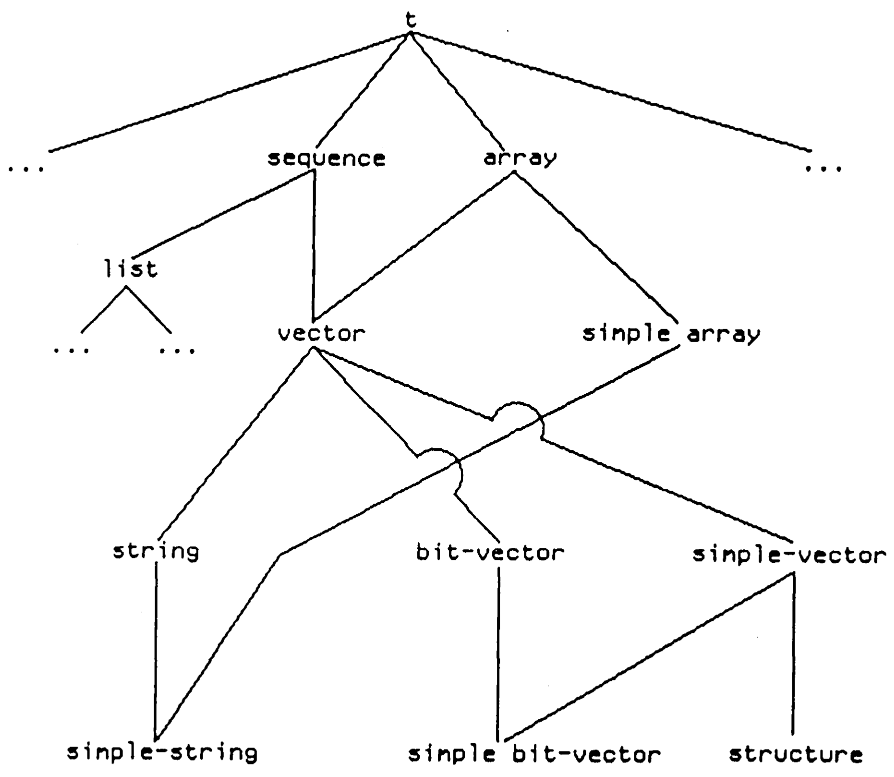


Figure 3. Symbolics Common Lisp Array Types

9.2 Basic Array Functions

SCL provides the following basic operations for arrays:

make-array	Creates and returns an array.
aref	Returns the specified element of the array.
setf	When used with **aref**, stores a value into the specified array element.
locf	When used with **aref**, returns a locative pointer to the specified element of the array.

These constants contain implementation-specific limits on arrays:

array-rank-limit	A constant that represents the upper exclusive bound on the rank of an array; the value is **8**.
array-dimension-limit	

A constant that represents the upper exclusive bound on each dimension of an array; the value is **134217728.**

array-total-size-limit
A constant that represents the upper exclusive bound on the total number of elements in an array; the value is **134217728.**

array-leader-length-limit
A constant that represents the upper exclusive bound on the length of an array leader; the value is **1024.**

Zetalisp provides the following basic operations for arrays:

zl:make-array	Creates and returns an array.
zl:aset	Stores a value into the specified array element. Note that setf of **aref** is preferred.
zl:aloc	Returns a locative pointer to the specified element of the array. Note that **locf** of **aref** is preferred.

For summaries of additional array operations: See the section "Common Operations on Arrays", page 177.

9.3 Creating Arrays

Use **make-array** to create new arrays.

make-array *dimensions* &key (*element-type* **t**) *initial-element* *initial-contents adjustable fill-pointer* *displaced-to displaced-index-offset* *displaced-conformally area leader-list* *leader-length named-structure-symbol* *Function*

make-array creates and returns a new array. *dimensions* is the only required argument. *dimensions* is a list of integers that are the dimensions of the array; the length of the list is the dimensionality, or rank of the array.

```
;; Create a two-dimensional array
(make-array '(3 4) :element-type 'string-char)
```

For convenience when making a one-dimensional array, the single dimension can be provided as an integer rather than a list of one integer.

```
;; Create a one-dimensional array of five elements.
(make-array 5)
```

The initialization of the elements of the array depends on the element type. By default the array is a general array, the elements can be any type of Lisp object, and each element of the array is initially **nil**. However, if the **:element-type** option is supplied, and it constrains the array elements to being integers or characters, the elements of the array are initially 0 or characters whose character code is 0 and style is **[nil.nil.nil]**. You can specify initial values for the elements by using the **:initial-contents** or **:initial-element** options.

Several of the keyword options are enhancements to Common Lisp. These include: **:displaced-conformally**, **:area**, **:leader-list**, **:leader-length**, and **:named-structure-symbol**.

See the section "Keyword Options For **make-array**", page 160.

See the section "Examples Of **make-array**", page 163.

9.3.1 Keyword Options For make-array

The keyword options for **make-array** can be any of the following:

:element-type

Enables you to specify the type of Lisp object allowed as elements of the array. The value should be a symbolic name of a type. The default type is **t**, which yields a general array that can contain elements of any type. For a list of allowed array types: See the section "Common Lisp Array Element Types", page 163.

The initialization of the elements of the array depends on the element type. If the array is of a type whose elements can only be integers or characters, the elements of the array are initially 0 or characters whose character code is 0 and style is **[nil.nil.nil]**. Otherwise, every element is initially **nil**.

To create a string, the **:element-type** option should be specified as **string-char** or **character**. Alternatively, you could use **make-string** instead of **make-array**. Note that if **:element-type** is **string**, this creates a general array, just as if **:element-type** were **t**. This is because the Symbolics computer does not have specialized arrays that hold just strings. See the section "Strings", page 221.

:initial-element

Initializes each element in the array to the supplied value. The value must be of the type specified by the **:element-type** argument, if that keyword was supplied. Example:

```
(make-array 5 :element-type 'string-char :initial-element #\a)
                                 => "aaaaa"
```

:initial-contents

Initializes the contents of the array. The value is a nested structure of sequences with values that correspond to the elements of the array. Example:

```
(make-array '(2 3 4) :initial-contents
            '(((a b c d) (1 2 3 4) (m n o p))
              ((e f g h) (5 6 7 8) (q r s t))))

=> #<ART-Q-2-3-4 34166170>
```

:adjustable

If not **nil**, specifies that the array's size can be altered dynamically after it has been created. The default is **nil**. The Genera implementation makes most arrays adjustable whether or not you use this option.

The following functions can be used to modify the size of an existing array:

adjust-array	Changes the size of an array.
zl:adjust-array-size	Resizes or reshapes the first dimension of an array. Note that **adjust-array** is preferred.
zl:array-grow	Creates a new array of the same type as the specified array, and forwards the old array to the new.

:fill-pointer

Specifies that the array should have a fill pointer and initializes the fill pointer to the value following the keyword. Note that **:fill-pointer** can only be used for one-dimensional arrays. Use this instead of **:leader-length** or **:leader-list** when you are using the leader only for a fill pointer. This argument defaults to **nil**. Fill pointers are discussed elsewhere: See the section "Array Leaders", page 164.

:displaced-to

Specifies that the array will be a *displaced* array, if the value is not **nil**. If the value is a fixnum or a locative, **make-array** creates a regular displaced array that refers to the specified section of virtual address space. If the value is an array, **make-array** creates an indirect array. See the section "Displaced Arrays", page 166. See the section "Indirect Arrays", page 166.

:displaced-index-offset

If this is present, the value of the **:displaced-to** option should be an array, and the value should be a nonnegative integer; it is made to be the index-offset of the created indirect or displaced array. See the section "Indirect Arrays", page 166.

The function **array-row-major-index** can aid in constructing the desired value for multidimensional arrays.

:displaced-conformally

Can be used with the **:displaced-to** option. If the value is **t** and **make-array** is creating an indirect array, the array uses conformal indirection. See the section "Conformal Indirection", page 168.

:area

The value specifies in which area the array should be created. It should be either an area number (an integer), or **nil** to mean the default area. This argument defaults to **nil**. See the section "Areas" in *Internals, Processes, and Storage Management.*

:leader-length

The value should be an integer. The array has a leader with that many elements. The elements of the leader are initialized to **nil** unless the **:leader-list**, **:fill-pointer**, or **:named-structure-symbol** option is given.

The leader-length must be less than **array-leader-length-limit**.

:leader-list

The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader are initialized from successive elements of this list. If the **:leader-length** option is not specified, then the length of the leader is *n*. If the **:leader-length** option is given, and its value is greater than *n*, then the extra leader elements are initialized to **nil**. If its value is less than *n*, an error is signalled. The leader elements are filled in forward order; that is, the **car** of the list is stored in leader element 0, the **cadr** in element 1, and so on. **:fill-pointer** overrides element 0, and **:named-structure-symbol** overrides element 1.

:named-structure-symbol

If this is not **nil**, it is a symbol to be stored in the named-structure cell of the array. The array is tagged as a named structure. See the section "Named Structures", page 343. If the array has a leader, then this symbol is stored in leader element 1 regardless of the value of the **:leader-list** option. If the array does not have a leader, then this symbol is stored in array element 0.

9.3.2 Common Lisp Array Element Types

This section lists the types that can be given as the **:element-type** option for **make-array**.

Element Type	*Contents of Array*
t	Any Lisp object
(unsigned-byte *n***)**	*n* is **1**, **2**, **4**, **8** or **16**. The array elements are positive integers limited in size to the number of bits indicated by *n*. Storing a larger fixnum, or a negative one, truncates it to the specified number of bits. Array elements are packed into 32-bit words.. If *n* is given as **1** and the array is one-dimensional, this is a bit-vector.
fixnum	Any fixnum, positive or negative.
character	Any character. If the array is one-dimensional, it is a fat string.
string-char	Characters in the Symbolics standard character set of character style NIL.NIL.NIL and bits field of zero. Array elements are packed four per word. If the array is one-dimensional, it is a thin string.
boolean	**t** or **nil**. Storing anything non-**nil** converts it to **t**. Elements are packed 32 per word.

9.3.3 Examples Of make-array

This section presents some examples of using **make-array**.

```
;; Create a one-dimensional array of five elements
(make-array 5)

;; Create a two-dimensional array
(make-array '(3 4))

;; Create an array with a three-element leader
(make-array 5 :leader-length 3)
```

```
;; Create an array of fixnums with a leader,
;; providing initial values for the leader elements
(setq a (make-array 100 :element-type 'fixnum
                        :leader-list '(t nil)))
(array-leader a 0) => T
(array-leader a 1) => NIL

;; Create a named-structure with five leader
;; elements, initializing some of them
(setq b (make-array 20 :leader-length 5
                       :leader-list '(0 nil foo)
                       :named-structure-symbol 'bar))
(array-leader b 0) => 0
(array-leader b 1) => BAR
(array-leader b 2) => FOO
(array-leader b 3) => NIL
(array-leader b 4) => NIL

;; Create a string with a fill pointer
(make-array 10 :element-type 'string-char
            :fill-pointer 5) => ""

;; Create a fat-string
(make-array 2 :element-type 'character
            :initial-element #\control-c)
```

9.4 Array Leaders

Any array can have an *array leader*. An array leader is similar to a one-dimensional general array that is attached to the main array. An array that has a leader acts like two arrays joined together. The leader can be stored into and examined with **setf** and **array-leader**. The leader is always one-dimensional and can always hold any kind of Lisp object, regardless of the type or dimensionality of the main part of the array. **array-leader-length-limit** is the exclusive upper bound on the length of an array leader, which is 1024 on the 3600.

Often the main part of an array is a homogeneous set of objects, while the leader is used to remember a few associated nonhomogeneous pieces of data. In this case the leader is not used like an array; each slot is used differently from the others. Explicit numeric subscripts should not be used for the leader elements of such an array; instead the leader should be described by using the **:array-leader** option to **defstruct**: See the macro **defstruct**, page 319.

By convention, element zero of the array leader of an array is used to hold the number of elements in the array that are "active" in some sense. When the zeroth element is used this way, it is called a *fill pointer*. Many array-processing functions recognize the fill pointer. For instance, if a string has seven elements, but its fill pointer contains the value 5, then only elements zero through four of the string are considered to be "active". This means that the string's printed representation is five characters long, string-searching functions stop after the fifth element, and so on.

The system does not provide a way to turn off the fill-pointer convention; any array that has a leader must reserve element 0 for the fill pointer or avoid using many of the array functions. If array leader element 0 contains a non-integer, such as **nil**, most functions act as if the array did not have a fill-pointer.

Leader element 1 is used in conjunction with the "named structure" feature to associate a "data type" with the array. See the section "Named Structures", page 343. Leader element 1 is treated specially only if the array is flagged as a named structure.

If there is no leader and the array is a named structure, the symbol goes in array element 0.

9.4.1 Operations on Array Leaders

The following functions are available for using with arrays that have leaders:

array-has-leader-p
Returns **t** if the given array has a leader; otherwise it returns **nil**.

array-leader Returns the specified element of the array leader. You can use **setf** and **locf** of **array-leader**.

array-leader-length
Returns the length of the given array's leader if it has one, or **nil** if it does not.

zl:ap-leader Returns a locative pointer to the specified element of the array leader. Note that **locf** of **array-leader** is preferred.

zl:store-array-leader
Stores a value in the specified element of the array leader. Note that **setf** of **array-leader** is preferred.

9.5 Displaced Arrays

Normally, an array is represented as a small amount of header information, followed by the contents of the array. However, sometimes it is desirable to have the header information removed from the actual contents. Such an array is known as a *displaced array*. One example of the usefulness of displaced arrays is when the contents of the array must be located in a special part of the Symbolics computer's address space, such as the area used for the control of input/output devices, or the bitmap memory that generates the TV image.

To create a displaced array, give **make-array** a fixnum or a locative as the value of the **:displaced-to** option. **make-array** creates a displaced array referring to that location of virtual memory and its successors.

References to elements of the displaced array access that part of storage, and return the contents. The normal array accessor functions (**aref** and **setf** with **aref**) are used on displaced arrays.

If the array is one whose elements are Lisp objects, caution should be used: if the region of address space does not contain typed Lisp objects, the integrity of the storage system and the garbage collector could be damaged. If the array is one whose elements are bytes, then there is no problem. It is important to know, in this case, that the elements of such arrays are allocated from the right to the left within the 32-bit words.

9.6 Indirect Arrays

It is possible to have an array whose contents, instead of being located at a fixed place in virtual memory, are defined to be those of another array. Such an array is called an *indirect array*, and is created by giving **make-array** an array as the value of the **:displaced-to** option.

The effects of this are simple if both arrays have the same type; the two arrays share all elements. An object stored in a certain element of one can be retrieved from the corresponding element of the other. This, by itself, is not very useful. However, if the arrays have different dimensionality, the manner of accessing the elements differs. Thus, by creating a one-dimensional array of nine elements that is indirected to a second, two-dimensional array of three elements by three, then the elements could be accessed in two different ways, either using **aref** on the one-dimensional array with one subscript; or using **aref** on the two-dimensional array with two subscripts.

To understand how the same element can be accessed two ways it is important to know that arrays are stored in row-major order in memory.

```
(setq a (make-array '(3 3) :initial-contents
                           '((one two three)
                             (four five six)
                             (seven eight nine))))

(setq b (make-array 9 :displaced-to a))

(aref b 0) => ONE
(aref a 0 0) => ONE

(aref b 1) => TWO
(aref a 0 1) => TWO

(aref b 6) => SEVEN
(aref a 2 0) => SEVEN
```

Unexpected effects can be produced if the new array is of a different type than the old array; this is not generally recommended. Indirecting an **art-*m*b** array to an **art-*n*b** array does the "obvious" thing. For instance, if *m* is 4 and *n* is 1, each element of the first array contains four bits from the second array, in right-to-left order.

9.6.1 Displaced and Indirect Arrays with Offsets

It is possible to create an indirect or displaced array in such a way that when an attempt is made to reference it or store into it, a constant number is added to the subscript given. This number is called the *index offset*, and is specified at the time the indirect array is created, by giving an integer to **make-array** as the value of the **:displaced-index-offset** option. Similarly, the length of the indirect array need not be the full length of the array it indirects to; it can be smaller. The **nsubstring** function creates such arrays. When using index offsets with multidimensional arrays, there is only one index offset; it is added in to the "linearized" subscript which is the result of multiplying each subscript by an appropriate coefficient and adding them together.

```
(setq a (make-array '(4 3)))

(setq b (make-array 5 :displaced-to a
                      :displaced-index-offset 2))
```

The second array is displaced to the first array. Also, the second array has an index offset of **2**. This affects the mapping of elements, which is illustrated below.

```
(aref b 0) is the same as (aref a 0 2)
(aref b 1) is the same as (aref a 1 0)
(aref b 2) is the same as (aref a 1 1)
(aref b 3) is the same as (aref a 1 2)
(aref b 4) is the same as (aref a 2 0)
```

9.6.2 Conformal Indirection

Multidimensional arrays remember their actual dimensions, separately from the coefficients by which to multiply the subscripts before adding them together to get the index into the array.

Multidimensional indirect arrays can have *conformal indirection.* If A is indirected to B, and they do not have the same number of columns, then normally the part of B that is shared with A does not have the same shape as A. If conformal indirection is used, the shape of array A changes. For example:

```
(setq b (make-array '(10. 20.)))
(setq a (make-array '(3 5) :displaced-to b
                           :displaced-index-offset
                             (array-row-major-index b 1 2)))
```

Now:

```
(aref a 0 1) = (aref b 1 3) and (aref a 1 1) = (aref b 1 8)
```

In contrast:

```
(setq a (make-array '(3 5) :displaced-to b
                           :displaced-index-offset
                             (array-row-major-index b 1 2)
                           :displaced-conformally t))
```

`(aref a 0 1) = (aref b 1 3)` still, but `(aref a 1 1) = (aref b 2 3)`. Each row of A corresponds to part of a row of B, always starting at the same column `(2)`.

A graphic illustration:

```
(setq a (make-array '(6 20.))
      b (make-array '(3 5) :displaced-to a
                           :displaced-index-offset
                              (array-row-major-index a 1 2))
      c (make-array '(3 5) :displaced-to a
                           :displaced-index-offset
                              (array-row-major-index a 1 2)
                           :displaced-conformally t))
```

```
  Normal case                    Conformal case
  0                  19          0                  19
  +--------------------+         +--------------------+
 0|aaaaaaaaaaaaaaaaaaaa|        0|aaaaaaaaaaaaaaaaaaaa|
  |aaBBBBBBBBBBBBBBBaaa|         |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|         |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|         |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|         |aaaaaaaaaaaaaaaaaaaa|
 5|aaaaaaaaaaaaaaaaaaaa|        5|aaaaaaaaaaaaaaaaaaaa|
  +--------------------+         +--------------------+
```

See the function **array-row-major-index** in *Symbolics Common Lisp: Language Dictionary*. See the section "Rasters", page 170.

The meaning of **adjust-array** for conformal indirect arrays is undefined.

All operations that treat a multidimensional array as if it were one-dimensional do not work on conformally displaced arrays:

copy-array-contents
copy-array-contents-and-leader
copy-array-portion
math:invert-matrix
zl:fillarray
zl:listarray

9.7 Vectors

A one-dimensional array is known as a *vector*. You can use the **:fill-pointer** option to **make-array** when making a vector, but not when making a multidimensional array. Several of the functions for vectors enable you to use the fill pointer capability of vectors.

A *general vector* allows its elements to be any type of Lisp object.

A *simple-vector* is a general vector that is not displaced, is not adjustable, and has no fill pointer. In Genera, the predicates such as **simple-vector-p** and **simple-bit-vector-p** can return **t** for vectors that are adjustable. Genera does not enforce the condition that a simple array must not be adjustable.

Bit-vectors are vectors that require their elements to be of type **bit**. SCL provides functions that operation on arrays of bits (which are not constrained to be bit vectors): See the section "Arrays of Bits", page 179.

Strings are vectors that require their elements to be of type **character** or **string-char**. Strings and string operations are described elsewhere: See the section "Strings", page 221.

9.7.1 Operations on Vectors

SCL provides the following functions for performing operations on vectors:

vector Creates a simple vector with specified initial contents.

array-has-fill-pointer-p
Tests whether the array has a fill pointer.

fill-pointer
Returns the value of the fill pointer.

sys:vector-bitblt
Copies a linear portion of one vector into a linear portion of another vector.

vector-push
Stores a new element in the element designated by the fill pointer and increments the fill pointer by one.

vector-push-extend
Like **vector-push**, but expands the vector if it is not large enough, and if the vector is adjustable.

vector-push-portion-extend
Copies a portion of one array to the end of another.

vector-pop
Decreases the fill pointer by one and returns the vector element designated by the new value of the fill pointer.

SCL provides the following predicate functions for determining if a given object is a vector, or a specialized vector:

vectorp	Tests whether the given object is a one-dimensional array.
simple-vector-p	Tests whether the given object is a simple vector.
bit-vector-p	Tests whether the given object is a one-dimensional array of bits.
simple-bit-vector-p	Tests whether the given object is a simple bit-vector.

9.8 Rasters

A raster is a two-dimensional array that is conceptually a two-dimensional rectangle of bits, pixels, or display items. Rasters are accessed in (x,y) fashion, rather than in (row,column) fashion. Rasters conceptually have width and height, while non-rasters have the numbers of columns and rows. In a row-major system, row corresponds to y and column corresponds to x; therefore a row of raster elements represents a row of the array.

Screen arrays, sheet arrays, bit arrays of the window system, fonts, and BFDs are rasters. Programs that access these items should use raster primitives rather than array primitives.

When using rasters, you should use **setf** to store into a raster element. Use **locf** to get a locative when the raster is a general array; **locf** is not allowed on arrays of bytes or of characters.

9.8.1 Operations on Rasters

The functions and methods for raster operations should be used only on rasters; they should not be used on non-rasters. User programs that provide an (x,y) style interface to rasters should use the raster functions to actually operate on the rasters.

bitblt	Copies a rectangular portion of one raster into a rectangular portion of another raster.
decode-raster-array	Returns various attributes of the raster as values.
make-raster-array	Makes rasters; takes keyword arguments that are accepted by **make-array**.
zl:make-raster-array	Makes rasters; takes keyword arguments that are accepted by **zl:make-array**.
raster-aref	Accesses the *x,y* graphics coordinates of a raster.
raster-index-offset	Returns a linear index of the array element of the specified coordinate.
raster-width-and-height-to-make-array-dimensions	Creates an argument that can be used to call **make-array**.
sys:page-in-raster-array	Ensures that the raster is in main memory.
sys:page-out-raster-array	Removes all the pages that represent the raster from main memory.

9.9 Planes

A *plane* is an array whose bounds, in each dimension, are minus-infinity and plus-infinity; all integers are valid as indices. Planes are distinguished not by size and shape, but by number of dimensions alone. When a plane is created, a default value must be specified. At that moment, every element of the plane has that value. As you cannot ever change more than a finite number of elements, only a finite region of the plane need actually be stored.

The regular array accessing functions do not work on planes. You can use **make-plane** to create a plane and **plane-aref** to get the value of an element. **setf** and **locf** work on **plane-aref**. **array-rank** works on a plane.

A plane is actually stored as an array with a leader. The array corresponds to a rectangular, aligned region of the plane, containing all the elements in which a **zl:plane-store** has been done (and others, in general, which have never been altered). The lowest-coordinate corner of that rectangular region is given by the **zl:plane-origin** in the array leader. The highest coordinate corner can be found by adding the **zl:plane-origin** to the **array-dimensions** of the array. The **plane-default** is the contents of all the elements of the plane that are not actually stored in the array. The **plane-extension** is the amount to extend a plane by in any direction when the plane needs to be extended. The default is 32.

If you never use any negative indices, then the **zl:plane-origin** is all zeroes and you can use regular array functions, such as **aref** and **zl:aset**, to access the portion of the plane which is actually stored. This can be useful to speed up certain algorithms. In this case you can even use the **2d-array-blt** function on a two-dimensional plane of bits or bytes, provided you don't change the **plane-extension** to a number that is not a multiple of 32.

9.9.1 Operations on Planes

The following functions are available for using with planes:

make-plane	Creates and returns a plane.
plane-aref	Returns the contents of a specified element of a plane. You can use **setf** and **locf** of **plane-aref**.
plane-default	Returns the contents of the infinite number of plane elements not actually stored.
plane-extension	Returns the amount to extend the plane by in any direction.
zl:plane-aset	Stores a datum into the specified element of a plane. **setf** of **plane-aref** is preferred.

zl:plane-origin	Returns the lowest coordinate values actually stored.
zl:plane-ref	Returns the contents of a specified element of a plane.
zl:plane-store	Stores a datum into the specified element of a plane.

9.10 Array Registers

The **aref**, **setf** of **aref**, and **zl:aset** operations on arrays consist of two parts:

1. They "decode" the array, determining its type, its rank, its length, and the address of its first data element.

2. They read or write the requested element.

The first part of this operation is not dependent on the particular values of the subscripts; it is a function only of the array itself.

When you write a loop that processes one or more arrays, the first part of each array operation is invariant if the arrays are invariant inside the loop. You can improve performance by moving this array-decoding overhead outside the loop, doing it only once at the beginning of the loop, rather than repeating it on every trip around the loop.

You can do this by using the **sys:array-register** and **sys:array-register-1d** declarations. **sys:array-register** is used for one-dimensional arrays, and **sys:array-register-1d** for multidimensional arrays. See the section "Function-body Declarations" in *Symbolics Common Lisp: Language Dictionary.*

9.10.1 Array Registers and Performance

The array-register feature makes optimization possible and convenient. Here is an example:

```
(defun foo (array-1 array-2 n-elements)
  (let ((a array-1)
        (b array-2))
    (declare (sys:array-register a b))
    (dotimes (i n-elements)
      (setf (aref b i) (aref a i)))))
```

This function copies the first **n-elements** elements of array **a** into array **b**. If the declaration is absent, it does the same thing more slowly. The variables **a** and **b** are compiled into "array register" variables rather than normal, local, variables. At the time **a** and **b** are bound, the arrays to which they are bound are decoded and the variables are bound to the results of the decoding. The compiler recognizes **aref** with a first argument that has been declared to be an array register, and **setf** of **aref** with a first argument that has been declared to be an array register; it compiles them as special instructions that do only the second part of the operation. These instructions are **fast-aref** and **fast-aset**.

If you want to verify that your array register declarations are working, follow these steps:

1. Compile the function.

2. Disassemble it: **(disassemble 'foo)**.

3. Look for **fast-aref** and **fast-aset** instructions. For example, note instructions 11 and 13:

```
 0  ENTRY: 3 REQUIRED, 0 OPTIONAL
 1  PUSH-LOCAL FP|0          ;ARRAY-1
 2  BUILTIN SETUP-1D-ARRAY TO 4       ;creating A(FP|3)
 3  PUSH-LOCAL FP|1          ;ARRAY-2
 4  BUILTIN SETUP-1D-ARRAY TO 4       ;creating B(FP|7)
 5  PUSH-LOCAL FP|2          ;N-ELEMENTS creating FP|11 (unnamed)
 6  PUSH-IMMED 0             ;creating I(FP|12)
 7  BRANCH 15
10  PUSH-LOCAL FP|12         ;I
11  FAST-AREF FP|4           ;A
12  PUSH-LOCAL FP|12         ;I
13  FAST-ASET FP|8           ;B
14  BUILTIN 1+LOCAL IGNORE FP|12      ;I
15  PUSH-LOCAL FP|12         ;I
16  PUSH-LOCAL FP|11
17  BUILTIN INTERNAL-< STACK
20  BRANCH-TRUE 10
21  RETURN-NIL
FOO
```

The performance advantage of array registers over the simplest types of array (for example, no leader or no displacement) is fairly small, since the normal **aref** and **zl:aset** operations on those arrays are quite fast. The real advantage of array registers is that they are equally fast for the more complicated arrays, such as indirect arrays and those with leaders, as they are for simple arrays.

The performance advantage to be gained through the use of array registers depends on the type of the array. Using an array register is never slower, except for one peculiar case: an indirect byte array with an index offset that is not a multiple of the number of array elements per word; in other words, an array whose first element is not aligned on a word boundary. An example of this case is:

```
(setq a (make-array 100 :element-type 'string-char))
(setq b (make-array 99 :element-type 'string-char
                       :displaced-to a
                       :displaced-index-offset 1))
```

If the **:displaced-index-offset** had been a multiple of 4, array registers would enhance performance.

9.10.2 Hints for Using Array Registers

The expansion of the **loop** macro's **array-elements** path copies the array into a temporary variable. In order to get the benefits of array registers, you must write code in the following way:

Right:

```
(defun tst1 (array incr)
  (declare (sys:array-register a))
  (loop for elt being the array-elements of array
              using (sequence a)
       sum (* elt incr)))
```

Wrong:

```
(defun tst (array incr)
  (let ((a array)) (declare (sys:array-register a))
       (loop for elt being the array-elements of a
            sum (* elt incr))))
```

loop generates a temporary variable; the "using" clause forces the temporary variable to be named **a**. Since the user gets to control the name of the variable, it is possible to assign a declaration to the variable.

The other way to do it is to avoid the **array-elements** path, and instead use:

```
(defun tst (array incr)
  (let ((a array)) (declare (sys:array-register a))
    (loop for i from 0 below (array-total-size a)
         sum (* (aref a i) incr))))
```

This is a bit more efficient because it does not have the overhead of setting up the variable **elt**.

9.10.3 Array Register Restrictions

It is not valid to declare a variable simultaneously to be **special** and to be **sys:array-register**. You cannot declare a parameter (a variable that appears in the argument-list of a **defun** or a **lambda**) to be an array register; you must bind another variable (perhaps with the same name) to it with **let** and declare that variable. For example:

```
(defun tst (x y)
  (let ((x x) (y y))
    (declare (sys:array-register x y))
    ...))
```

An array-register variable cannot be a free lexical variable; it must be bound in the same function that uses it.

Note that the **array-register** declaration is in the **system** package (also known as **sys**), and therefore the declaration is **sys:array-register** or **sys:array-register-1d**. Be sure to type **sys:array-register** and not just **array-register** to gain compile-time advantages such as checking for misspelled declarations. Also, if you type **array-register**, the code generated by the compiler runs slower. Note that if you type **sys:array-regist*a*r** instead of the correct spelling, the package system catches the misspelling because the **system** package is locked.

If the array decoded into an array register is altered (for example, with **adjust-array**) after the array register is created, the next reference through the array register re-decodes the array.

9.11 Matrices and Systems of Linear Equations

Matrices are represented as two-dimensional Lisp arrays. These functions that operate on matrices are part of the mathematics package rather than the kernel array system, hence the "**math:**" in the names.

math:decompose and **math:solve** are used to solve sets of simultaneous linear equations. **math:decompose** 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 only need to call **math:decompose** once. In terms of their argument names, these two functions exist to solve the vector equation $A\ x = b$ for x. A is a matrix. b and x are vectors.

9.11.1 Operations on Matrices

The following functions perform some useful matrix operations:

math:decompose	Computes the LU decomposition of a matrix.
math:determinant	Returns the determinant of a matrix.
math:fill-2d-array	Stores values into elements into an array.
math:invert-matrix	Computes the inverse of a matrix.
math:list-2d-array	Returns a list of lists contains the values in an array.
math:multiply-matrices	Multiplies one matrix by another matrix.
math:solve	Solves a set of simultaneous equations.
math:transpose-matrix	Transposes a matrix.

9.11.2 Common Operations on Arrays

Several summary tables of array operations appear elsewhere in the documentation. These include:

See the section "Basic Array Functions", page 158.
See the section "Operations on Array Leaders", page 165.
See the section "Operations on Planes", page 172.
See the section "Operations on Rasters", page 171.
See the section "Operations on Vectors", page 170.
See the section "Operations on Matrices", page 177.

9.11.2.1 Getting Information About an Array

The following functions can be used to get information about arrays:

array-dimension	Returns the length of the specified dimension of an array.
array-dimensions	Returns a list whose elements are the dimensions of the array.
array-has-leader-p	Returns **t** if the given array has a leader; otherwise it returns **nil**.

array-in-bounds-p	Tests whether a subscript is valid for the array.
array-leader-length	Returns the length of an array's leader.
length	Returns the number of elements in a vector (or list). For vectors with a fill pointer, the active length as specified by the fill pointer is returned.
array-rank	Returns the number of dimensions of an array.
array-row-major-index	Returns an integer that identifies the accessed element.
array-total-size	Returns the number of elements in an array.
array-element-type	Returns the type of the elements of an array.
adjustable-array-p	Returns **t** if the array is adjustable.
sys:array-row-span	Returns the number of array elements spanned by one of its rows.
zl:array-active-length	Returns the active number of elements in an array; **length** provides the same functionality.
zl:array-dimension-n	Returns the size for the specified dimension of the array.
sys:array-displaced-p	Tests whether the array is a displaced array.
array-has-leader-p	Tests whether the array has a leader.
sys:array-indexed-p	Tests whether an array is an indirect array with an index-offset.
sys:array-indirect-p	Tests whether an array is indirect.
zl:array-length	Returns the number of elements in an array.
zl:array-#-dims	Returns the dimensionality of an array; **array-rank** provides the same functionality.

9.11.2.2 Changing the Size of an Array

The following functions can be used to modify the size of an existing array:

adjust-array	Changes the size of an array.
zl:adjust-array-size	Resizes or reshapes the first dimension of an array. Note that **adjust-array** is preferred.
zl:array-grow	Creates a new array of the same type as the specified array, and forwards the old array to the new.

9.11.2.3 Arrays of Bits

The following functions are available for use with arrays of bits:

bit	Returns the specified element of an array of bits.
sbit	Returns the specified element of a simple array of bits.
bit-and	Performs logical *and* operations on the specified arrays.
bit-ior	Performs logical *inclusive or* operations on the specified arrays.
bit-xor	Performs logical *exclusive or* operations on the specified arrays.
bit-eqv	Performs logical *exclusive nor* operations on the specified elements.
bit-nand	Performs logical *not and* operations on the specified elements.
bit-nor	Performs logical *not or* operations on the specified elements.
bit-not	Takes an array as an argument and returns a copy of the array with all the bits inverted.
bit-andc1	Performs logical *and* operations on the complement of *argument1* with *argument2.*
bit-andc2	Performs logical *and* operations on *argument1* with the complement of *argument2.*
bit-orc1	Performs logical *or* operations on the complement of *argument1* with *argument2.*
bit-orc2	Performs logical *or* operations on the complement of *argument2* with *argument1.*
bit-vector-p	Tests whether the given object is a one-dimensional array of bits.

9.11.2.4 Adding to the End of an Array

The following functions can be used to add to the end of an array:

vector-pop	Decreases the fill pointer by one and returns the vector element designated by the new value of the fill pointer.
vector-push	Stores a new element in the element designated by the fill pointer and increments the fill pointer by one.
vector-push-extend	Like **vector-push**, but expands the vector if it is not large enough, and if the vector is adjustable.
vector-push-portion-extend	Copies a portion of one array to the end of another.
zl:array-pop	Decreases the fill pointer by one.
zl:array-push	Store an object in a specified element of an array and increases the fill pointer by one.
zl:array-push-extend	Stores an object in a specified element of an array, growing the array if necessary.
zl:array-push-portion-extend	Copies a portion of one array to the end of another.

9.11.2.5 Copying an Array

The following functions can be used to copy the contents of arrays:

2d-array-blt	Copies a rectangular portion of one array into a portion of another array; intended for two dimensional arrays.
bitblt	Copies a rectangular portion of one raster into a rectangular portion of another raster.
copy-array-contents-and-leader	Copies the contents and leader of one array into the contents of another array.
copy-array-contents	Copies the contents of one array into the contents of another array.
copy-array-portion	Copies a portion of the contents of one array into the contents of another array.

list-array-leader	Creates and returns a list whose elements are those of the leader of the specified array.
replace	Copies contents of one vector (or list) into another.
zl:fillarray	Fills up an array with the elements of a list.
zl:listarray	Creates and returns a list of the elements of the specified array.

9.11.2.6 Accessing Multidimensional Arrays as One-dimensional

The **sys:array-register-1d** declaration is used together with the following functions to access multidimensional arrays as if they were one-dimensional. See the section "Function-body Declarations" in *Symbolics Common Lisp: Language Dictionary.*

This declaration allows loop optimization of multidimensional array subscript calculations. The user must do the reduction from multiple subscripts to a single subscript.

For an example: See the function **sys:%1d-aref** in *Symbolics Common Lisp: Language Dictionary.*

sys:%1d-aref	Returns the specified array element; like **aref** except that it acts as if the array were one-dimensional.
sys:%1d-aset	Stores a value into the specified array element; like **zl:aset** except that it acts as if the array were one-dimensional.
sys:%1d-aloc	Returns a locative pointer to the array element-cell selected by the index; like **zl:aloc** except that it acts as if the array were one-dimensional.
sys:array-row-span	Returns the number of array elements spanned by one of its rows.

9.11.2.7 Array Representation Tools

The following functions and variables are primitives.

sys:*array-type-codes*	A variable that is a list of all the array type symbols.
sys:array-bits-per-element	An association list that associates array type symbols with size.
sys:array-bits-per-element	A function that returns the number of bits per cell for unsigned numeric arrays.

sys:array-element-size
A function that returns the number of bits that fit into an element of the specified array.

sys:array-elements-per-q
An association list that associates each array type symbol with the number of array elements stored in one word.

sys:array-elements-per-q
A function that returns the number of array elements stored in one word.

sys:array-types A function that returns the symbolic name of the array type.

9.11.2.8 Other Array Functions

sys:return-array Attempts to return an array to free storage. This is a subtle and dangerous feature.

9.11.3 Row-major Storage of Arrays

This section describes how arrays are stored in memory. This is an implementation detail that does not concern most programmers. However, if you use some of the advanced array practices, such as displaced arrays or adjusting the array size dynamically, you need to understand how arrays are stored in memory.

Genera stores multi-dimensional arrays in *row-major* order. The following 2 by 3 two-dimensional array illustrates row-major order. Two-dimensional arrays have rows and columns. The number of rows is the span of the first dimension and the number of columns is the span of the second dimension. When accessing a two-dimensional array, the row is the first subscript and the column is the second subscript.

	Column		
	0	*1*	*2*
Row			
0	0,0	0,1	0,2
1	1,0	1,1	1,2

In row-major order, the array elements are arranged in memory in the following order:

```
(0,0) (0,1) (0,2) (1,0) (1,1) (1,2)
```

In other words, the sequence is determined by going across the row from column to column. Thus, the first, or row, index remains constant while the second, or column, index changes as you follow the linear sequence in memory.

9.11.4 Performance and Arrays

In Genera 7.0, arrays are stored in row-major order rather than column-major order, which was used in Release 6.1 and all previous releases. The change from column-major to row-major order affects paging performance when large arrays are used. To reference every element in a multidimensional array and move linearly through memory to improve locality of reference in Genera 7.0, you must vary the last subscript fastest. In 6.1 and earlier releases, vary the first subscript fastest.

Certain programs might run faster or slower due to the interaction of multidimensional array storage and paging. For example, a Release 6 database might be organized as 4 by *n*. The 4 items associated with the *i*th object are at consecutive array locations and are therefore likely on the same page. This is because of Release 6's column-major order. In Genera 7.0, the elements (0,*i*), (1,*i*), (2,*i*) and (3,*i*) are not consecutive and can be on different pages. It might be desirable to change all aspects of the database to be *n* by 4 so that (*i*,0), (*i*,1), (*i*,2), and (*i*,3) are consecutive. A useful technique for hiding some of this is to define substs that take care of the 0,1,2,3 values.

For example, in a column-major system you might have the following:

```
(defsubst item-name  (table index) (aref table 0 index))
(defsubst item-color (table index) (aref table 1 index))
```

In a row-major system you can have the same names and contracts with different implementations:

```
(defsubst item-name  (table index) (aref table index 0))
(defsubst item-color (table index) (aref table index 1))
```

9.11.5 Compatibility Operations for Arrays

9.11.5.1 Zetalisp Array Types

This section describes the Zetalisp array types. Zetalisp array types are known by a set of symbols whose names begin with "**art-**" (for ARray Type). For example, a general array is called a Zetalisp **sys:art-q** array. Zetalisp has many types of specialized arrays, such as **sys:art-fixnum** and **sys:art-boolean**. This terminology is being phased out in favor of Common Lisp terminology.

sys:art-q Array Type

The most commonly used type is called **sys:art-q**. A **sys:art-q** array simply holds Lisp objects of any type. This array type can store single-precision floating-point numbers without any storage overhead.

sys:art-q-list Array Type

Similar to the **sys:art-q** type is **sys:art-q-list**. Its elements can be any Lisp object. The difference is that a **sys:art-q-list** array "doubles" as a list; the function **g-l-p** takes a **sys:art-q-list** array and returns a list whose elements are those of the array, and whose actual substance is that of the array. If you either **rplaca** the elements of the list or **setf** the **car** of a sublist, the corresponding element of the array changes, and if you store into the array, the corresponding element of the list changes the same way. An attempt to either **rplacd** the list or **setf** the **cdr** of a sublist causes an error, since arrays cannot implement that operation.

The following function manipulates **sys:art-q-list** arrays:

g-l-p	Returns a list that stops at the fill pointer.

You cannot use **make-array** to create a **sys:art-q-list** array. If you need to create such an array, use **zl:make-array**.

sys:art-*N*b Array Type

There is a set of types called **sys:art-1b**, **sys:art-2b**, **sys:art-4b**, **sys:art-8b**, and **sys:art-16b**. These names are short for "1 bit", "2 bits", and so on. Each element of a **sys:art-*n*b** array is a nonnegative integer, and only the least significant *n* bits are remembered in the array; all of the others are discarded. Thus **sys:art-1b** arrays store only 0 and 1, and if you store a 5 into a **sys:art-2b** array and look at it later, you find a 1 rather than a 5.

These arrays are used when it is known beforehand that the integers that are stored are nonnegative and limited in size to a certain number of bits. Their advantage over the **sys:art-q** array is that they occupy less storage, because more than one element of the array is kept in a single machine word. (For example, 32 elements of a **sys:art-1b** array or 2 elements of a **sys:art-16b** array fits into one word).

sys:art-string Array Type

A **sys:art-string** array type is an array whose elements are simple characters. One-dimensional arrays of this type are character strings.

sys:art-fat-string Array Type

A **sys:art-fat-string** array is a string whose elements are *fat characters*. For a description of fat strings: See the section "Introduction to Strings", page 221.

sys:art-boolean Array Type

A **sys:art-boolean** array type is an array whose elements can take on the values **t** and **nil**. It uses only one bit of storage per element.

sys:art-fixnum Array Type

sys:art-fixnum is an array type that stores fixnums only. It is similar to the **sys:art-1b**, **sys:art-2b** through **sys:art-16b** array types except that **sys:art-fixnum** arrays can also store negative fixnums. In contrast, **sys:art-*n*b** arrays always store the low *n* bits and return positive fixnums when read.

For example, the following example creates a square, 2-dimensional array of fixnums with 1024 elements on a side:

```
(make-array '(1024 1024) :element-type 'fixnum)
```

locf and **zl:aloc** are invalid on **sys:art-fixnum** arrays, as that would provide a means to store something other than a fixnum into the array.

sys:art-fixnum arrays are similar to **sys:art-q** arrays except that storing a non-fixnum signals an error. **sys:art-fixnum** arrays can be used as the array arguments to **bitblt** and **2d-array-blt** arrays (as can **sys:art-q** arrays whose elements are fixnums), and the error checking ensures all the entries are fixnums. They can also be used for disk-arrays.

9.11.5.2 Zetalisp Array Accessing Primitives

New programs should use the basic array functions: **aref**, **setf** of **aref**, and **locf** of **aref**. There is no reason for any program to call the array primitives **zl:ar-1**, **zl:as-1**, **zl:ar-2**, and so forth explicitly. These primitives are documented because many old programs use them.

The compiler turns **aref** into **zl:ar-1** and **zl:ar-2** according to the number of subscripts specified. It also turns **zl:aset** into **zl:as-1** and **zl:as-2** and **zl:aloc** into **zl:ap-1** and **zl:ap-2**.

9.11.5.3 Maclisp Array Compatibility

The functions in this section are provided only for Maclisp compatibility, and should not be used in new programs.

Integer arrays and flonum-only arrays do not exist. "Un-garbage-collected" arrays do not exist.

Readtables and obarrays are represented as arrays, but unlike Maclisp special array types are not used. Information about readtables and obarrays (packages) can be found elsewhere: See the function **zl:read** in *Reference Guide to Streams, Files, and I/O*. See the function **zl:intern** in *Symbolics Common Lisp: Language Dictionary*. There are no "dead" arrays, nor are Multics "external" arrays provided.

Subscripts are always checked for validity, regardless of the value of **zl:*rset** and whether the code is compiled or not.

For information about putting arrays into binary files: See the section "Putting Data in Compiled Code Files" in *Reference Guide to Streams, Files, and I/O*.

The **zl:*rearray** function is not provided, since not all of its functionality is available in Genera. The most common uses can be replaced by **adjust-array**.

Arrays can be treated as functions. When this is done, each array has a name, which is a symbol whose function definition is the array. Genera supports this style by allowing an array to be *applied* to arguments, as if it were a function. The arguments are treated as subscripts and the array is referenced appropriately. Note that this method is very slow and inefficient, and is considered poor style.

In Maclisp, arrays are usually kept on the **array** property of symbols, and the symbols are used as function names. In order to provide some degree of compatibility for this manner of using arrays, the **array** and **zl:*array** functions are provided, and when arrays are applied to arguments, the arguments are treated as subscripts and **apply** returns the corresponding element of the array.

The functions that exist for Maclisp compatibility are:

zl:*array	Creates an **sys:art-q** type array.
array	Creates an **sys:art-q** type array.
zl:arraydims	Returns a list whose first element is the type of array and whose remaining elements are its dimensions.

10. Sequences

10.1 Introduction to Sequences

A *sequence* is a type that contains an ordered set of elements. It embraces both lists and vectors (one-dimensional arrays).

Depending on your specific application, you might choose to represent ordered sets as lists or strings. Symbolics Common Lisp provides generic sequence functions that operate on both lists or vectors. These functions perform basic operations on sequences of Lisp objects, irrespective of the underlying representation of the sequence. It makes sense to reverse a sequence or extract a range of sequence elements, whether the sequence is implemented as a vector or a list. The following sequence functions are defined in Symbolics Common Lisp:

concatenate	copy-seq	count	count-if
count-if-not	delete	delete-duplicates	delete-if
delete-if-not	elt	every	fill
find	find-if	find-if-not	length
make-sequence	map	merge	mismatch
notany	notevery	nreverse	nsubstitute
nsubstitute-if	nsubstitute-if-not	position	position-if
position-if-not	reduce	remove	remove-duplicates
remove-if	remove-if-not	replace	reverse
search	some	sort	stable-sort
subseq	substitute	substitute-if	substitute-if-not

Zetalisp has analogous functions for some of these operations:

zl:delete	zl:every	zl:length	zl:map
zl:nreverse	zl:remove	zl:reverse	zl:some
zl:sort	zl:stable-sort		

Some of these functions have variants formed by a prefix or a suffix, for example, **reverse** and **nreverse**, and **position**, **position-if**, and **position-if-not**.

In addition, many functions accept keyword arguments that modify the sequence operations.

10.2 How the Reader Recognizes Sequences

The reader does not recognize a sequence as such; it recognizes its component types, lists and vectors.

See the section "How the Reader Recognizes Conses", page 131.

A vector can be denoted by surrounding its components by #(and), as in #(a b c). The most common kind of vector is a string. A string is a vector whose elements are characters. The reader knows that a string is being entered when it receives a sequence of characters surrounded by double quotes ("). See the section "How the Reader Recognizes Strings", page 223.

10.3 Type Specifiers and Type Hierarchy for Sequences

The type specifiers relating to sequences are as follows:

array	vector	list	symbol
simple-array	bit-vector	cons	null
simple-vector	simple-bit-vector		keyword
structure			

Details about each type specifier appear in its dictionary entry.

Figure 4 shows the relationships between the various data types relating to sequences. For more on data types, type specifiers, and type-checking in Symbolics Common Lisp: See the section "Data Types and Type Specifiers", page 69.

10.4 Sequence Operations

The sequence operations fall into six major categories:

- Construction and access
- Predicates
- Mapping
- Modification, including
    - Reducing
    - Replacing
- Searching
- Sorting and merging

Whenever a sequence function constructs a returns a new vector, it always returns a simple vector; similarly any strings constructed will be simple strings.

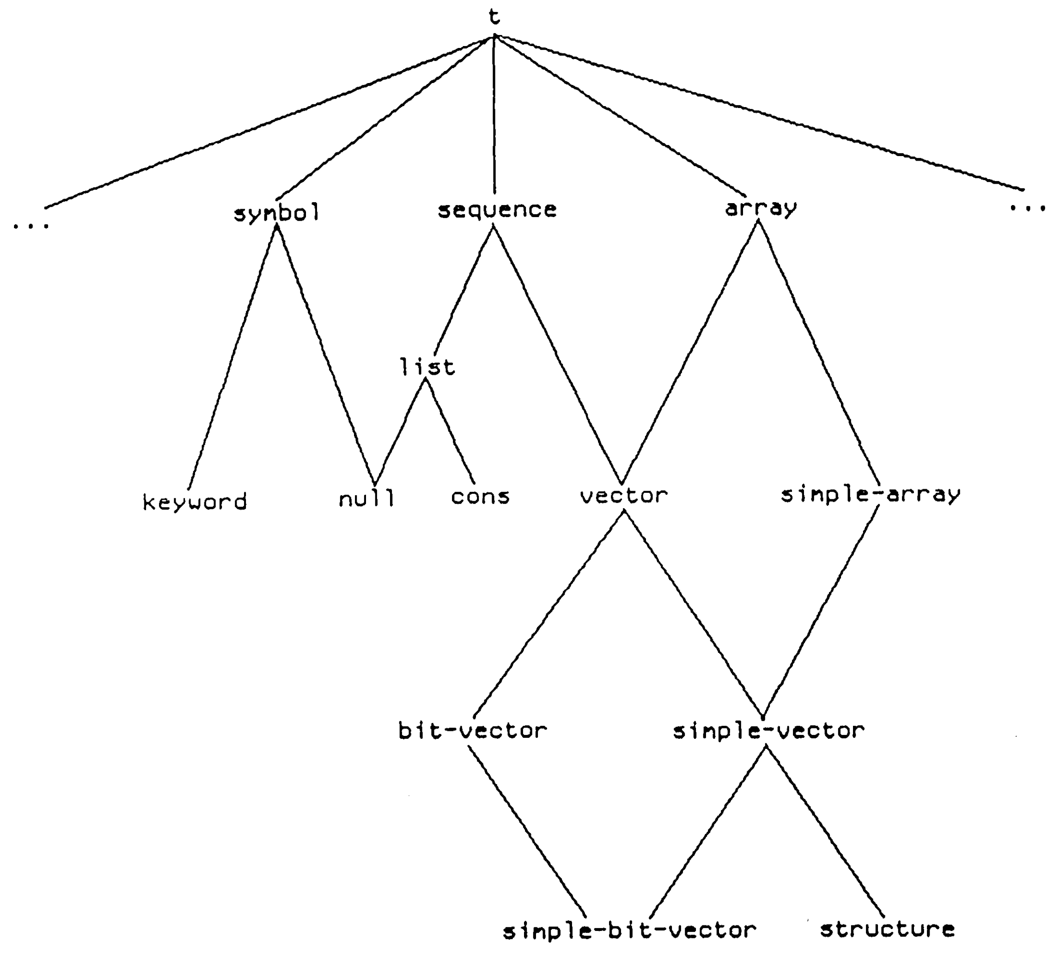


Figure 4. Symbolics Common Lisp Sequence Data Types

The sequence functions accept a number of keyword arguments. For the sake of efficiency, some of these arguments delimit and direct sequence operations. These keywords include the following:

> **:start**
> **:end**
> **:start1, :start2**
> **:end1, :end2**
> **:from-end**
> **:count**

These arguments are explained in the appropriate dictionary entries. Other keyword arguments, including **:test**, **:test-not**, and **:key**, allow you to selectively perform operations on the elements of a sequence according to some stated criterion.

10.4.1 Testing Elements of a Sequence

Elements of a sequence can be tested either by using the appropriate keyword (**:test**, **:test-not**, **:key**) or by using one of the **-if** or **-if-not** variants of the basic sequence operations (for example, **remove**, **remove-if**, **remove-if-not**).

If an operation requires testing elements of the sequence according to some criterion, then the criterion can be specified in one of the following ways:

- The operation accepts an item argument, and sequence elements are tested for being **eql** to *item*. (Note: **eql** is the default test.) For example, the **remove** operation returns a copy of *sequence* from which all elements **eql** to *item* have been removed.

  ```
  (remove item sequence)
  ```

- The variants formed by appending **-if** and **-if-not** to the function name accept a one-word argument predicate (not an item), and sequence elements are tested for satisfying and not satisfying the predicate. For example, the **remove-if** operation returns a copy of *sequence* from which all numbers have been removed.

  ```
  (remove-if #'numberp sequence)
  ```

- The operation accepts the **:test** or **:test-not** keywords, which allow you to specify a test other than the default, **eql**. (Note: it is not valid to use both **:test** and **:test-not** in the same call.) For example, the **remove** operation returns a copy of *sequence* from which all elements **equal** to *item* have been removed.

  ```
  (remove item sequence :test #'equal)
  ```

- You can modify sequence elements before they are passed to the testing function by using the **:key** keyword argument. In this way you can create arbitrarily complicated tests for operating on sequences. **:key** takes a function of one argument that will extract from an element the part to be tested in place of the whole (original) element. For example, the lambda expression decrements each element in the vector before the element is tested for being **eql** to 0.

  ```
  (delete 0 #(1 2 1) :key #'(lambda (x) (- x 1))) => #(2)
  ```

  Another example: the find operation searches for the first element of *sequence* whose car is **eq** to *item*.

  ```
  (find item sequence :test #'eq :key #'car)
  ```

In the sequence operations that require a test, an element x of a sequence satisfies the test if any of the following conditions is true. *keyfn* is the value of the **:key** keyword argument, whose default is the identity function.

- A basic function is called, *testfun* is specified by **:test**, and `(funcall` *testfun item* `(`*keyfn x*`))` is true.
- A basic function is called, *testfun* is specified by **:test-not**, and `(funcall` *testfun item* `(`*keyfn x*`))` is false.
- An **-if** function is called, and `(funcall` *predicate* `(`*keyfn x*`))` is true.
- An **-if-not** function is called, and `(funcall` *predicate* `(`*keyfn x*`))` is false.

Similarly, two elements x and y of a sequence match if either of the following is true.

- *testfun* is specified by **:test**, and `(funcall` *testfun* `(`*keyfn x*`)` `(`*keyfn y*`))` is true.
- *testfun* is specified by **:test-not**, and `(funcall` *testfun* `(`*keyfn x*`)` `(`*keyfn y*`))` is false.

The order in which arguments are given to *testfun* corresponds to the order in which those arguments (or the sequence containing those arguments) were passed to the sequence function in question. If a sequence function gives two elements from the same sequence argument to *testfun*, the elements are passed in the same order in which they appear in the sequence.

10.4.2 Sequence Construction and Access

The following functions perform simple operations on sequences. **make-sequence**, **concatenate**, and **copy-seq** create new sequences. Whenever a sequence function constructs and returns a new vector, that vector is always a simple vector; any new strings returned will be simple strings.

elt *sequence index*	Extracts an element from the *sequence* at position *index*. Returns that element.
subseq *sequence start &optional end*	Non-destructively creates a subsequence of the argument *sequence*. Returns a new sequence.
copy-seq *sequence &optional area*	Non-destructively copies *sequence*. Returns a new *sequence* which is **equalp** (not **eq**) to *sequence*.
concatenate *result-type &rest sequences*	Combines the elements of the *sequences* in the order the *sequences* were given as arguments. Returns the new combined sequence.

length *sequence*	Counts the number of elements in *sequence*. Returns a non-negative integer.
make-sequence *type size &key initial-element area*	Creates a sequence. Returns a sequence.
zl:length *x*	Counts the number of elements in the list *x*. Returns a non-negative integer. Use Common Lisp equivalent **length**.

10.4.3 Predicates That Operate on Sequences

The predicates take as many arguments as there are sequences provided. *predicate* is first applied to the elements with index 0 in each of the sequences, and perhaps then to the elements with index 1, and so on, until a criterion for termination is met or the end of the shortest of the *sequences* is reached.

Note that **zl:some** and **zl:every** have analogies in Symbolics Common Lisp.

some *predicate &rest sequences*	Each element in *sequence* is tested against *predicate*. Returns whatever value *predicate* returns as non-**nil**, as soon as any invocation of *predicate* returns a non-**nil** value. Otherwise returns **nil**.
every *predicate &rest sequences*	Each element in *sequence* is tested against *predicate*. Returns **nil** as soon as any invocation of *predicate* returns **nil**. Otherwise returns non-**nil**.
notany *predicate &rest sequences*	Each element in *sequence* is tested against *predicate*. Returns **nil** as soon as any invocation of *predicate* returns a non-**nil** value. Otherwise returns non-**nil**.
notevery *predicate &rest sequences*	Each element in *sequence* is tested against *predicate*. Returns non-**nil** as soon as any invocation of *predicate* returns a **nil** value. Otherwise returns **nil**.
zl:some *list pred &optional (step #'cdr)*	Each element in *list* is tested against *pred*. Returns a tail of *list* such that the car of the tail is the first element that the *predicate*

returns non-**nil** when applied to, or **nil** if *predicate* returns **nil** for every element.

zl:every *list pred &optional (step #'cdr)*
Each element, default *step*, in the *list* is tested against the *pred*. Returns **t** if *predicate* returns non-**nil** when applied to every element of *list*, or **nil** if *predicate* returns **nil** for some element.

10.4.4 Mapping Sequences

Mapping is a type of iteration in which a function is successively applied to pieces of one or more sequences. The result is a sequence containing the respective results of the function applications. **map** can be applied to any kind of sequence, but the other map-type functions operate only on lists. **reduce** is included here because of its conceptual relationship to mapping.

map *result-type function &rest sequences*
Applies *function* to *sequences*. Returns a new sequence such that element *i* of the new sequence is the result of applying *function* to element *i* of each of the argument *sequences*.

reduce *function sequence &key from-end (start 0) end (initial-value nil initial-value-p)*
Combines the elements of the *sequence* using a binary operation. Returns the result of using the *function* on the *sequence*.

zl:map *fcn list &rest more-lists*
Applies *fcn* to *list* and to successive sublists of that list. Returns a new list such that element *i* of the new list is the result of applying *function* to element *i* of each of the argument *lists*.

10.4.5 Sequence Modification

Each of these modifying operations alters the contents of a sequence or produces an altered copy of a given sequence. Some of these functions have separate "destructive" versions, prefixed by the letter "n", for example, **nreverse**. Others have **-if** and **-if-not** variants of the basic sequence operation. Many of the searching functions accept the testing keywords: **:test**, **:test-not**, and **:key**.

Note that certain Zetalisp operations have analogous functions in Symbolics Common Lisp.

reverse *sequence*	Reverses the elements of the *sequence*. Returns a new reversed sequence.
nreverse *sequence*	Destructive version of **reverse**. Returns a modified sequence.
zl:reverse *list*	Creates a new list whose elements are the elements of *list* taken in reverse order. Returns a new list.
zl:nreverse *l*	Reverses a list *l*. Returns a reversed list.

10.4.5.1 Reducing Sequences

remove *item sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end count*

Non-destructively removes occurrences of *item* in the *sequence*. Returns a new sequence.

remove-if *predicate sequence &key key from-end (start 0) end count*

Non-destructively removes those items from the *sequence* which satisfy *predicate*. Returns a new sequence.

remove-if-not *predicate sequence &key key from-end (start 0) end count*

Non-destructively removes those items from the *sequence* which do not satisfy the *predicate*. Returns a new sequence.

delete *item sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end count*

Destructive version of **remove**. Returns a modified sequence.

delete-if *predicate sequence &key key from-end (start 0) end count*

Destructive version of **remove-if**. Returns a modified sequence.

delete-if-not *predicate sequence &key key from-end (start 0) end count*
Destructive version of **remove-if-not**. Returns a modified sequence.

remove-duplicates *sequence &key from-end (test #'eql) test-not (start 0) end key*
Non-destructively removes duplicate elements from *sequence*. Returns a new sequence.

delete-duplicates *sequence &key (test #'eql) test-not (start 0) end from-end key replace*
Destructive version of **remove-duplicates**. Returns a modified sequence.

zl:delete *item list &optional (ntimes -1)*
Deletes occurrences of *item* in the *list* (**equal** is used for the comparison). Returns the *list* with all occurrences of *item* removed. Use **delete** instead.

zl:remove *item list &optional (times most-positive-fixnum)*
Non-destructive version of *zl:delete*. Use **remove** instead.

10.4.5.2 Replacing Sequences

fill *sequence item &key (start 0) end*
Destructively replaces each element of the *sequence* with *item*. Returns the modified sequence.

replace *sequence1 sequence2 &key (start1 0) end1 (start2 0) end2*
Destructively modifies *sequence1* by copying into it successive elements from *sequence2*.

substitute *newitem olditem sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end count*
Non-destructively replaces *newitem* for *olditem* in *sequence*. Returns a new sequence.

substitute-if *newitem predicate sequence &key key from-end (start 0) end count)*
Non-destructively replaces *newitem* for elements in *sequence* which satisfy *predicate*. Returns a new sequence.

substitute-if-not *newitem predicate sequence &key key from-end (start 0) end count*
Non-destructively replaces *newitem* for elements in *sequence* which do not satisfy *predicate*. Returns a new sequence.

nsubstitute *newitem olditem sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end count*
Destructive version of **substitute**. Returns a modified sequence.

nsubstitute-if *newitem predicate sequence &key key from-end (start 0) end count*
Destructive version of **substitute-if**. Returns a modified sequence.

nsubstitute-if-not *newitem predicate sequence &key key from-end (start 0) end count*
Destructive version of **substitute-if-not**. Returns a modified sequence.

10.4.6 Searching for Sequence Items

Each of the searching functions searches a sequences to locate one or more elements satisfying some test.

find *item sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end*
Finds the leftmost *item* in the *sequence*. Returns *item* if found, else **nil**.

find-if *predicate sequence &key key from-end (start 0) end*
Finds the leftmost element in *sequence* which satisfies *predicate*. Returns said element if found, else **nil**.

find-if-not *predicate sequence &key key from-end (start 0) end*
Finds the leftmost element in *sequence* which does not satisfy *predicate*. Returns said element if found, else **nil**.

position *item sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end*
Finds the leftmost *item* in the *sequence*. Returns the index of the item if found, else **nil**.

position-if *predicate sequence &key key from-end (start 0) end*
Finds the leftmost element in the *sequence* satisfying the *predicate*. Returns the index of the element if found else **nil**.

position-if-not *predicate sequence &key key from-end (start 0) end*
Finds the leftmost element in the *sequence* not

satisfying *predicate*. Returns the index of the element if found else **nil**.

count *item sequence &key (test #'eql) test-not (key #'identity) from-end (start 0) end*
Counts the number of elements in the specified subsequence of *sequence* satisfying the predicate specified by **:test** keyword. Returns a non-negative integer.

count-if *predicate sequence &key key from-end (start 0) end*
Counts the number of elements in the specified subsequence of *sequence* satisfying the *predicate*. Returns a non-negative integer.

count-if-not *predicate sequence &key key from-end (start 0) end*
Counts the number of elements in the specified subsequence of *sequence* that do not satisfy the *predicate*. Returns a non-negative integer.

mismatch *sequence1 sequence2 &key from-end (test #'eql) test-not key (start1 0) (start2 0) end1 end2*
Compares the specified subsequences of *sequence1* and *sequence2* element-wise. Returns **nil** if they are of equal length and match at every element. Otherwise, the result is a non-negative integer representing where the sequences failed to match.

search *sequence1 sequence2 &key from-end (test #'eql) test-not key (start1 0) (start2 0) end1 end2*
Looks for a subsequence of *sequence2* that element-wise matches *sequence1*. Returns **nil** if no such subsequence exists. Otherwise, it returns the index into *sequence2* of the leftmost element of the leftmost such matching subsequence.

10.4.7 Sorting and Merging Sequences

The sorting and merging functions destructively modify argument sequences in order to place a sequence into a sorted order or to merge two previously sorted sequences.

sort *sequence predicate &key key*	Destructively modifies *sequence* by sorting it according to an order determined by *predicate*. Returns a modified sequence.
stable-sort *sequence predicate &key key*	Same as **sort**, however **stable-sort** guarantees that elements considered equal by *predicate* will remain in their original order.
merge *result-type sequence1 sequence2 predicate &key key*	Destructively merges the *sequence1* and *sequence2* according to an order determined by *predicate*. Returns merged sequences.
zl:sort *x sort-lessp-predicate*	Sorts the contents of the array or list *x* by the order determined by *sort-lessp-predicate*. Returns a modified list or array *x*. Use the **sort** Common Lisp equivalent.
zl:stable-sort *x lessp-predicate*	Same as **zl:sort**, however **zl:stable-sort** guarantees that elements considered equal by *predicate* will remain in their original order. Use the **stable-sort** Common Lisp equivalent.

11. Characters

For an introduction to characters: See the section "Overview of Characters", page 24.

11.1 How the Reader Recognizes Characters

The reader recognizes characters by the #\ prefix followed by the character's name. For example:

#\A	*is read as the character*	A
#\1	*is read as the character*	1
#\Space	*is read as the character*	Space
#\control-A	*is read as the character*	c-A

The following examples show how to represent the character A with various bits set:

Meta bit:	#\meta-A or #\m-A
Hyper bit:	#\hyper-A or #\h-A
Super bit:	#\super-A or #\s-A
Control bit:	#\control-A or #\c-A
Control and meta bits:	#\c-m-A or #\m-c-A
All bits set:	#\h-s-m-c-A (or other combinations)

The reader recognizes characters that are in character sets other than the Symbolics character set by the #\ prefix followed by the name of the character set, a colon, and the name of the character. For example:

#\mouse:nw-arrow	nw-arrow character in mouse character set
#\mouse:scissors	scissors character in mouse character set
#\arrow:eye	eye character in arrow character set

11.2 Type Specifiers and Type Hierarchy for Characters

Characters are Lisp objects of type **character**. **character** has two subtypes: **string-char** and **standard-char**.

character All characters are of type **character**.

string-char	This is a subtype of **character**. Characters that are in the Symbolics standard character set with bits field of zero and style of NIL.NIL.NIL are of type **string-char**.
standard-char	This is a subtype of **string-char**. Characters that are in the Common Lisp standard character set are of type **standard-char**.

The Common Lisp standard character set includes:

```
! " # $ % & ' ( ) * + , - . / 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 { | } ~
```

Genera also supports the following semi-standard Common Lisp characters:

#\Backspace #\Tab #\Linefeed #\Page #\Return #\Rubout

Genera calls any character that is of type **string-char** a *thin character* because it can be represented with less storage space. A character that is not of type **string-char** because it is in a character set other than the Symbolics character set, or because it contains non-zero bits or style information is called a *fat character.*

For a complete list of characters supported in the Symbolics standard character set: See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O.*

For a list of character type-checking predicates: See the section "Character Predicates", page 218.

11.3 Character Objects

A *character* is a type of Lisp object. A character object is used to represent letters of the alphabet and numbers, among other things. Characters are the building blocks of strings; a string is a one-dimensional array of characters.

Each character object has the following attributes: the character code, the character set, the bits, and the character style.

Character code	Identifies this character, such as "uppercase A".
Character style	Specifies how the character should appear. For example: FIX.ROMAN.NORMAL
Bits	Indicates whether any of these bits are set for the character: Control, Meta, Super, and Hyper.

11.3.1 Fields of a Character

The following diagram depicts the fields of a character:

```
|<-----------------Entire character---------------->|
|<---Bits--->|<--Style--->|<-Char set->|<-Subindex->|
                          |<-------Char code------->|
```

This diagram makes it clear that a character object is composed of three independent attributes: the bits, the character style, and the character code. The character code can be broken down into the character set and a subindex into that character set.

Genera provides functions that access the various fields of a character:

Function	*Field accessed*
char-int	Entire character
char-code	Character code field
char-bits	Bits field
sys:char-subindex	Subindex field
si:char-style	Returns the character style object that is associated with the integer stored in the Style field.

There is a one-to-one correspondence between each character style (such as NIL.NIL.NIL and SWISS.BOLD.NORMAL) and the character style index, which is the integer stored in the style field. This association is maintained in a system table, and it is different from one machine to another, and can be different when you cold boot your machine. Do not write programs that depend on a character style index representing the same character style from one cold boot to another, or from one machine to another.

Common Lisp has a font field instead of a character style field. As implemented in SCL, characters have no font field and the **char-font-limit** is **1**. This is in compliance with Common Lisp.

In Symbolics documentation the word font is used in two contexts: to describe a font that is specific to a device, for representing characters; to refer to the font of a character as implemented in releases of Symbolics software prior to Genera 7.0.

11.3.2 Character Sets

The *code* field of a character can be broken down into a character set and an index into that character set.

A *character set* is a set of related characters that are recognizably different from other characters. Genera supports the standard Symbolics character set, which is an upward-compatible extension of the 96 Common Lisp standard characters and the 6 Common Lisp semi-standard characters. It is nearly an upward-compatible extension of ASCII; it uses a single Newline character and omits the ASCII control characters. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O.*

Another example of a character set is the Cyrillic alphabet; this is not implemented in Genera.

When comparing characters, there is no intrinsic ordering between characters in different character sets. Two characters of different character sets are never equal. Less-than is not well defined between them. Within a single character set, less-than is defined so that characters (and strings) can be sorted alphabetically.

Genera supports three character sets: the Symbolics standard character set; the mouse character set, and the arrow character set. Figure 5 shows the characters in the mouse character set. Figure 6 shows the characters in the mouse character set.

Characters that are in character sets other than the Symbolics character set are represented by the #\ prefix followed by the name of the character set, a colon, and the name of the character. For example:

```
#\mouse:nw-arrow
#\mouse:scissors
#\mouse:trident
#\arrow:center-dot
#\arrow:eye
#\arrow:circle-cross
```

11.3.3 Character Code, Bits, and Style

The character code is a field of the character that identifies that character. Other systems use an ASCII code to identify a character. Characters that are recognizably distinct always have different character codes. For example, the Roman a and the Greek α have two different character codes.

A character can be modified by the bits field and the character style field. These two modifications of a character leave it recognizably the same; it does not change the character code.

The bits field describes whether the hyper, super, control, or meta key is part of this character. The character #\A has character code 65 and a bits field of 0. The character #\c-A also has character code 65, but the bits field is set to **char-control-bit**, which means that this is a control character. For a list of constants that represent the control, hyper, super, and meta bits: See the section "Character Bit Constants", page 220.

The character style describes how a character should appear. For example, the Roman a, the bold **a**, and the italic *a* all have the same character code. The style

Up-Arrow	↑	NE-Arrow	↗
Right-Arrow	→	Circle-Times	⊗
Down-Arrow	↓	Big-Triangle	▶
Left-Arrow	←	Medium-Triangle	▶
Vertical-Double-Arrow	↕	Small-Triangle	▸
Horizontal-Double-Arrow	↔	Inverse-Up-Arrow	⍐
NW-Arrow	↖	Inverse-Down-Arrow	⍗
Times	×	Filled-Lozenge	⬥
Fat-Up-Arrow	⬆	Dot	
Fat-Right-Arrow	➡	Fat-Times	✕
Fat-Down-Arrow	⬇	Small-Filled-Circle	•
Fat-Left-Arrow	⬅	Filled-Circle	●
Fat-Double-Vertical-Arrow	⬍	Fat-Circle	○
Fat-Double-Horizontal-Arrow	⬌	Fat-Circle-Minus	⊖
Paragraph	¶	Fat-Circle-Plus	⊙
NW-Corner	┌	Down-Arrow-To-Bar	⤓
SE-Corner	┘	Short-Down-Arrow	↓
Hourglass	⌛	Up-Arrow-To-Bar	⤒
Circle-Plus	⊕	Short-Up-Arrow	↑
Paintbrush	🖌	Boxed-Up-Triangle	⍓
Scissors	✂	Boxed-Down-Triangle	⍌
Trident	Ψ		

Figure 5. Mouse Character Set

Center-Dot	•	Right-Arrowhead-Dot	⋗	Left-Arrowhead	<
Circle-Plus	⊕	Left-Arrowhead-Dot	⋖	Right-Arrowhead	>
Circle-Cross	⊗	Right-Triangle	▶	Right-Open-Arrow	⇒
Down-Arrowhead	v	Up-Open-Arrow	⬆	Baseline-Caret	▴
Up-Arrowhead	ʌ	Right-Hand	☞	Right-Short-Open-Arrow	▸
Right-Fat-Arrow	➽	Left-Hand	☜	Open-X	×
		Eye	◂		

Figure 6. Arrow Character Set

field also expresses such attributes of a character as its displayed size and the typeface used.

An operational definition of the difference between the *code* and *style* fields is provided by the **char-equal** function, which compares the character code and bits but ignores the style.

11.3.4 eq and Character Objects

Instead of using **eq** on character objects, use **char-equal** or **char=**. **char=** compares characters exactly, according to all fields including code, bits, character style, and alphabetic case. **char-equal** compares characters according to their code and bits, ignoring case and character style.

eq is not well defined on character objects. Changing a field of a character object gives you a "new copy" of the object; it never modifies somebody else's "copy" of "the same" character object. In this way character objects are similar to integers with fields accessed by **ldb** and changed by **dpb**. Because **eq** is not well defined on character objects, you should use **eql** to compare characters for identity, not the **eq** function. Currently on the 3600 family of machines, **eq** and **eql** are equivalent for characters, just as they are equivalent for integers, but programs should not be written to depend on this, for two reasons:

- "Extended" character objects could be introduced in the future, standing in the same relationship to "basic" character objects as bignums do to fixnums.
- **eq** might not work for characters in other implementations of the Common Lisp dialect.

11.4 Character Styles

11.4.0.1 What is a Character Style?

A *character style* is a combination of three characteristics that describe how a character appears. These characteristics are the *family*, *face*, and *size*.

Family	Characters of the same family have a typographic integrity, so that all characters of the same family resemble one another. Examples: SWISS, DUTCH, and FIX.
Face	A modification of the family, such as BOLD or ITALIC.
Size	The size of the character, such as NORMAL or VERY-SMALL.

The character style is the grouping of the family, face, and size fields. A character style is often represented by the convention:

family.face.size

An example of a fully specified character style is:

```
SWISS.ITALIC.LARGE
```

Each element of the character style can be specified or left unspecified. A family, face, or size of NIL means to use the default value. Most characters have the following character style:

```
NIL.NIL.NIL
```

Characters of style NIL.NIL.NIL are displayed in the default character style established for the current output device.

11.4.1 Default Character Styles

The appearance of a character depends on two things: the character style of the character, and the default character style. The default character style is a global parameter of an output device. It applies for all processes. Windows, buffers, files, and printers have each have default character styles for output. The default character style specifies the appearance of a character whose character style is NIL.NIL.NIL. The character's style is merged against the default character style to produce the final appearance of the character. A default character style must be fully specified.

We recommend that you use character styles by making good use of the default character styles. You preserve the most flexibility by keeping the character style of the characters themselves as unspecified as possible. If you want to change the appearance of all characters in a Zmacs buffer, a Zmail message or a window, you can change the default character style instead of changing the character style of each character.

The default character style affects the appearance of a character on output. There is also a typein character style for each interactive stream, which is normally NIL.NIL.NIL. The typein character style affects the character style in which characters are entered as input. If the typein character style is NIL.BOLD.NIL, any characters you enter at the keyboard have the character style NIL.BOLD.NIL. It is important to be sure that the application program can handle characters whose character style is something other than NIL.NIL.NIL, if you are going to use a typein character style other than NIL.NIL.NIL.

If you only want to change the way that characters echo, but not the way they are entered as input, you can change the echo character style. See the section "Using Character Styles in the Input Editor" in *User's Guide to Symbolics Computers.*

11.4.2 Merging Character Styles

This section gives some examples of how the character style of a character is merged against the default character style to produce a final result.

In general, we advise that you specify as little as possible when changing a character style. That is, if you want the character's face to be italic, specify only the face component and let the family and size come from the default character style.

Character Style of a Character	*Default Character Style*	*Result of Merging*
NIL.NIL.NIL	FIX.ROMAN.NORMAL	FIX.ROMAN.NORMAL
NIL.ITALIC.LARGE	FIX.ROMAN.NORMAL	FIX.ITALIC.LARGE
NIL.ITALIC.SMALLER	FIX.ROMAN.NORMAL	FIX.ITALIC.SMALL
SWISS.BOLD.LARGER	FIX.ROMAN.NORMAL	SWISS.BOLD.LARGE

The family and face components are either NIL or the name of a family or face.

The size component can be NIL, an *absolute size* (such as LARGE or VERY-SMALL) or a *relative size* (such as LARGER or SMALLER). A relative size is merged against the default size such that when you merge LARGER against NORMAL, the result is the next size larger than NORMAL.

The ordered hierarchy of sizes is:

TINY
VERY-SMALL
SMALL
NORMAL
LARGE
VERY-LARGE
HUGE

If you try to merge SMALLER against the smallest size, TINY, the result is TINY. Similarly, if you try to merge LARGER against the largest size, HUGE, the result is HUGE.

11.4.3 Available Character Styles

This section lists the most commonly used character families, faces, and sizes. For a mapping of each style to a specific font for the black and white console or the LGP2 printer: See the section "Character Styles for TV Fonts", page 214. See the section "Character Styles for LGP2 Fonts", page 215.

The following families, faces, and sizes are available and are mapped to fonts in all combinations, for the black and white console and the LGP2 printer.

Families	DUTCH, SWISS, FIX
Faces	ROMAN, BOLD, ITALIC, BOLD-ITALIC
Sizes	VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE

The black and white console device supports these additional character styles:

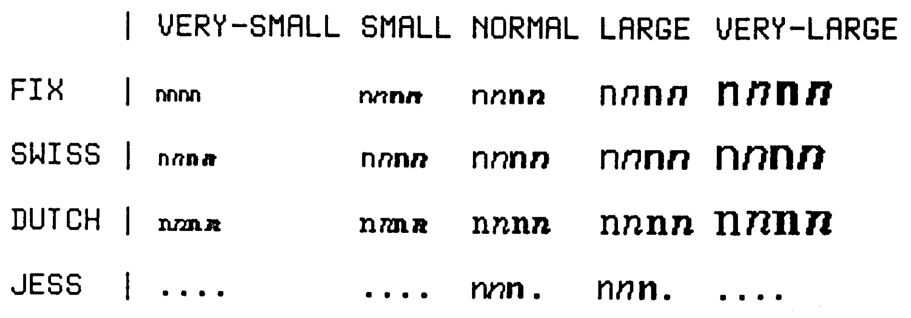
- The family JESS in all combinations of faces ROMAN, ITALIC, BOLD, and sizes NORMAL and LARGE.
- The family EUREX in face ITALIC and size HUGE.

The LGP2 printer device supports this additional character style:

- The family HEADING in all combinations of faces ROMAN, ITALIC, BOLD, BOLD-ITALIC, and sizes VERY-SMALL, SMALL, NORMAL, LARGE, and VERY-LARGE.

The following figures show how a character appears in different families, faces, and sizes. This output came from a black-and-white screen, so it displays TV fonts.

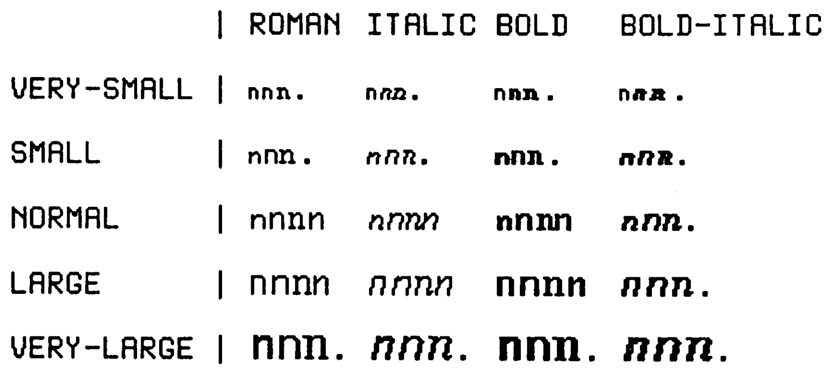
Figure 7 shows how the characters of the five most common sizes appear for a given family and face. The sizes of the displayed characters from left to right are: VERY-SMALL, SMALL NORMAL, LARGE, VERY-LARGE.



```
       | VERY-SMALL  SMALL  NORMAL  LARGE  VERY-LARGE

FIX    | nnnn        nnnn   nnnn    nnnn   nnnn

SWISS  | nnnn        nnnn   nnnn    nnnn   nnnn

DUTCH  | nnnn        nnnn   nnnn    nnnn   nnnn

JESS   | ....        ....   nnn.    nnn.   ....
```

Figure 7. Varying Character Sizes: VERY-SMALL to VERY-LARGE

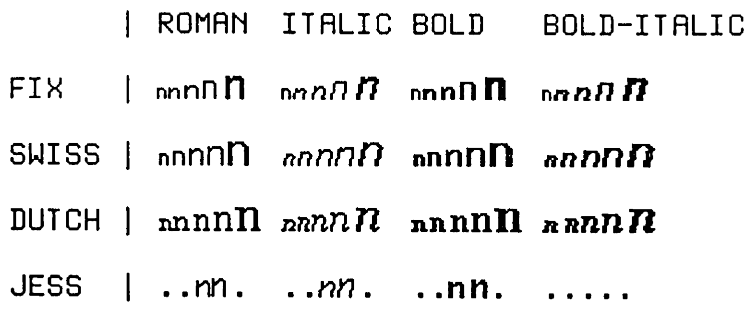
Figure 8 shows how the characters of the four most common faces appear for a given family and size. The faces of the displayed characters from left to right are: ROMAN, ITALIC, BOLD, and BOLD-ITALIC.



```
           | ROMAN  ITALIC BOLD   BOLD-ITALIC
VERY-SMALL | nnn.   nnn.   nnn.   nnn .
SMALL      | nnn.   nnn.   nnn.   nnn.
NORMAL     | nnnn   nnnn   nnnn   nnn.
LARGE      | nnnn   nnnn   nnnn   nnn.
VERY-LARGE | nnn.   nnn.   nnn.   nnn.
```

Figure 8. Varying Character Faces: ROMAN, ITALIC, BOLD, BOLD-ITALIC

Figure 9 shows how the characters of the four most common families appear for a given face and size. The families of the displayed characters from top to bottom are: FIX, SWISS, DUTCH, and JESS.



```
       | ROMAN  ITALIC BOLD   BOLD-ITALIC
FIX    | nnnnn  nnnnn  nnnnn  nnnnn
SWISS  | nnnnn  nnnnn  nnnnn  nnnnn
DUTCH  | nnnnn  nnnnn  nnnnn  nnnnn
JESS   | ..nn.  ..nn.  ..nn.  .....
```

Figure 9. Varying Character Families: FIX, SWISS, DUTCH, JESS

11.4.4 Using Character Styles

Genera offers facilities for using character styles to specify how a character should appear. You can use commands in Zmacs, Zmail, and the input editor to change the character style of a character, or to change the default character style associated with a buffer, mail message, or window. Similarly, you can change the default character style associated with a printer, for a particular print request.

Refer to the following sections for descriptions of facilities for using character styles:

In *User's Guide to Symbolics Computers* see:

"Using Character Styles in the Input Editor"
"Character Styles and the Lisp Listener"
"Using Character Styles in Zmail"
"Using Character Styles in Hardcopy"

In *Text Editing and Processing* see:

"Using Character Styles in Zmacs"

In *Programming the User Interface* see:

with-character-style
with-character-family
with-character-face
with-character-size

11.4.5 Mapping a Character Style to a Font

A character style is device-independent. However, when a character is displayed on a device, somehow a specific font must be chosen to represent the character. The final appearance of the character depends on: the character code, the character set, the character style, and the device.

The associations between character styles and fonts that are specific to a device are contained in **si:define-character-style-families** forms. Genera 7.0 provides associations for a large set of the pre-defined character styles to fonts on the most commonly used devices, such as the console and the LGP2 and LGP1 printers.

You can use **si:get-font** to determine which font is chosen for a given device, character set, and character style.

If you want to use a private font, you can either use it via device fonts, or use **si:define-character-style-families** to explicitly associate one or more character styles with that font. Using **si:define-character-style-families** has the advantage of hooking the new font into the character style system, but it has the disadvantage that any user who reads in a file using the newly defined character style must already have that style defined in the world. Using device fonts has the advantage that users can conveniently read in files that use private fonts (there is no need to have a form defining new character styles). The disadvantages of device fonts are: they circumvent the character style system and they are not device-independent. That is, a device font can work for characters to be displayed on the screen, or on some other device, but not both.

si:get-font *device character-set style* &optional (*error-p* **t**) *inquiry-only* *Function*

Given a *device, character-set* and *style,* **si:get-font** returns a font object that would be used to display characters from that character set in that style on the device. This is useful for determining whether there is such font mapping for a given device/set/style combination.

If *error-p* is non-**nil**, this function signals an error if no font is found. If *error-p* is **nil** and no font isfound, **si:get-font** returns **nil**.

If *inquiry-only* is provided, the returned value is not a font object, but some other representation of a font, such as a symbol in the **fonts** package (for screen fonts) or a string (for printer fonts).

```
(si:get-font si:*b&w-screen* si:*standard-character-set*
             '(:jess :roman :normal))

=> #<FONT JESS13 154102066>

(si:get-font lgp:*lgp2-printer* si:*standard-character-set*
             '(:swiss :roman :normal) nil t)

=> "Helvetica10"
```

si:define-character-style-families *device character-set* &rest *plists* *Function*

This function is the mechanism for defining new character styles, and for defining which font should be used for displaying characters from *character-set* on the specified *device*. *plists* contain the actual mapping between character styles and fonts.

It is necessary that a character style be defined in the world before you access a file that uses the character style. You should be careful not to put any characters from a style you define into a file that is shared by other users, such as sys.translations.

It is possible for *plists* to map from a character style into another character style; this usage is called *logical character styles*. It is expected that the logical style used has its own mapping, in this **si:define-character-style-families** form or another such form, that eventually is resolved into an actual font.

plists is a nested structure whose elements are of the form:

```
(:family family
         (:size size
                (:face face target-font
                 :face face target-font
                 :face face target-font)
          :size size
                (:face face target-font
                 :face face target-font)))
```

Each *target-font* is one of:

- A symbol such as **fonts:cptfont**, which represents a font for a black and white Symbolics console.

- A string such as **"furrier7"**, which represents a font for an LGP2 printer.
- A list whose **car** is **:font** and whose **cadr** is an expression representing a font, such as `(:font ("Furrier" "B" 9 1.17))`. This is also a font for an LGP2 printer.
- A list whose **car** is **:style** and whose **cdr** is a character style, such as: `(:style` *family face size*`)`. This is an example of using a logical character style (see ahead for more details).

Each *size* is either a symbol representing a size, such as **:normal**, or an asterisk * used as a wildcard to match any size. The wildcard syntax is supported for the **:size** element only. When you use a wildcard for size the *target-font* must be a character style. The size element of *target-font* can be **:same** to match whatever the size of the character style is, or **:smaller** or **:larger**.

If you define a new size, that size cannot participate in the merging of relative sizes against absolute sizes. The ordered hierarchy of sizes is predefined. See the section "Merging Character Styles", page 206.

The elements can be nested in a different order, if desired. For example:

```
(:size size
       (:face face
              (:family target-font)))
```

The first example simply maps the character style BOX.ROMAN.NORMAL into the font **fonts:boxfont** for the character set **si:*standard-character-set*** and the device **si:*b&w-screen***. The face ROMAN and the size NORMAL are already valid faces and sizes, but BOX is a new family; this form makes BOX one of the valid families.

```
;;; -*- Package:SYSTEM-INTERNALS; Mode:LISP; Base: 10 -*-

(define-character-style-families *b&w-screen* *standard-character-set*
  '(:family :box
    (:size :normal (:face :roman fonts:boxfont))))
```

Once you have compiled this form, you can use the Zmacs command Change Style Region (invoked by `c-X c-J`) and enter BOX.ROMAN.NORMAL. This form does not make any other faces or sizes valid for the BOX family.

The following example uses the wildcard syntax for the **:size**, and associates the faces **:italic, :bold**, and **:bold-italic** all to the same character style of BOX.ROMAN.NORMAL. This is an example of using logical character styles. This form has the effect of making several more character styles valid; however, all styles that use the BOX family are associated with the same logical character style, which uses the same font.

```
;;; -*- Package:SYSTEM-INTERNALS; Mode:LISP; Base: 10 -*-

(define-character-style-families *b&w-screen* *standard-character-set*
  '(:family :box
    (:size * (:face :italic (:style :box :roman :normal)
                    :bold (:style :box :roman :normal)
                    :bold-italic (:style :box :roman :normal)))))
```

For lengthier examples: See the section "Examples Of **si:define-character-style-families**", page 212.

11.4.5.1 Examples Of si:define-character-style-families

The use and syntax of **si:define-character-style-families** is best explained by example.

The following example maps character styles for the standard Symbolics character set (which is bound to **si:*standard-character-set***) on the device **si:*b&w-screen***:

```
;;; -*- Package:SYSTEM-INTERNALS; Mode:LISP; Base: 10 -*-

(define-character-style-families *b&w-screen* *standard-character-set*
  '(:family :fix
    (:size :normal (:face :roman fonts:cptfont
                          :italic fonts:cptfonti
                          :bold fonts:cptfontcb
                          :bold-italic fonts:cptfontbi
                          :bold-extended fonts:cptfontb
                          :condensed fonts:cptfontc
                          :extra-condensed fonts:cptfontcc)
           :small (:face :roman fonts:tvfont
                         :italic fonts:tvfonti
                         :bold fonts:tvfontb
                         :bold-italic fonts:tvfontbi)
           :very-small (:face :roman fonts:einy7
                              :italic fonts:einy7
                              :bold fonts:einy7
                              :bold-italic fonts:einy7
                              :uppercase fonts:5x5)
           :tiny (:face :roman fonts:tiny
                        :italic fonts:tiny
```

```
                    :bold fonts:tiny
                    :bold-italic fonts:tiny)
     :large (:face :roman fonts:medfnt
                    :italic fonts:medfnti
                    :bold fonts:medfntb
                    :bold-italic fonts:medfntbi)
     :very-large (:face :roman fonts:bigfnt
                         :italic fonts:bigfnti
                         :bold fonts:bigfntb
                         :bold-italic fonts:bigfntbi))))
```

The following example maps character styles for the standard Symbolics character set (which is bound to **si:*standard-character-set***) on the device **lgp:*lgp2-printer***:

```
;;; -*- Package:SYSTEM-INTERNALS; Mode:LISP; Base: 10 -*-

(define-character-style-families lgp:*lgp2-printer*
                                 *standard-character-set*
  '(:family :fix
    (:size  :small (:face :roman "Furrier7"
                          :italic "Furrier7I"
                          :bold "Furrier7B"
                          :bold-italic "Furrier7BI")
            :normal (:face :roman "Furrier9"
                           :italic "Furrier9I"
                           :bold "Furrier9B"
                           :bold-extended (:font ("Courier" "B" 9 1.17))
                           :bold-italic "Furrier9BI")
            :large (:face :roman "Furrier11"
                          :italic "Furrier11I"
                          :bold "Furrier11B"
                          :bold-italic "Furrier11BI"))))
```

11.4.5.2 Device Fonts

This section describes the facility for using device fonts to display characters. If you use device fonts you circumvent the character style system; device fonts ignore the default character style of the output device, and no merging is supported for them. Unlike character styles, device fonts are not device independent. If a character is displayed in a device font, it cannot be displayed on two different devices. For example, if a character is in a device font intended for the screen, it cannot be hardcopied.

The main reason for using device fonts is to compensate for a possible problem in using **si:define-character-style-families**. Suppose you define new character styles using **si:define-character-style-families** and write a file that contains the newly defined character styles. If anyone else reads that file, it is necessary that the character styles have already been defined in that world, by virtue of the **si:define-character-styles** form having been evaluated in that world.

In contrast, if you use device fonts to specify how characters appear in the file, and the font is stored in the sys:fonts;tv;*.*.* directory, other users can can read the file, and characters appear in the correct font. Note that you cannot hardcopy that file because the characters in the screen device font cannot be directed to another device such as an LGP2 printer. We strongly discourage using device fonts in electronic mail. If the device font is intended for the black and white console, the message cannot be hardcopied.

A secondary reason for using device fonts is for convenience when developing fonts intended for the screen. You can simply display characters in the new font by using device fonts, and skip the step of defining character styles for the font until you are ready to do so.

To use device fonts, you use character style commands and enter DEVICE-FONT as the family. You are then prompted for the name of the font, which must be a symbol in the **font** package.

For example, in Zmacs, when you use `c-X c-J` to change the style of a region, you can enter DEVICE-FONTS for the family. You can then press `HELP` for a list of fonts defined for the screen. Choose one of the fonts. There is no need to enter a size. The characters are then displayed in the chosen device font.

Two presentation types also accept device fonts: **character-face-or-style** and **character-style-for-device**.

11.4.5.3 Character Styles for TV Fonts

This section shows the mapping from a character style to a font for the black and white console device. Each family has its own table. The rows are the various faces and the columns are the sizes. If no font is available for a *family.face.size* triple, "--" is shown in that spot.

Family FIX

	TINY	VERY-SMALL	SMALL	NORMAL	LARGE	VERY-LARGE
ROMAN	TINY	EINY7	TVFONT	CPTFONT	MEDFNT	BIGFNT
ITALIC	TINY	EINY7	TVFONTI	CPTFONTI	MEDFNTI	BIGFNTI
BOLD	TINY	EINY7	TVFONTB	CPTFONTCB	MEDFNTB	BIGFNTB
BOLD-ITALIC	TINY	EINY7	TVFONTBI	CPTFONTBI	MEDFNTBI	BIGFNTBI
UPPERCASE	--	5X5	--	--	--	--
BOLD-EXTENDED	--	--	--	CPTFONTB	--	--

CONDENSED	--	--	--	CPTFONTC	--	--
EXTRA-CONDENSED	--	--	--	CPTFONTCC	--	--

Family SWISS

	VERY-SMALL	SMALL	NORMAL	LARGE	VERY-LARGE
ROMAN	HL8	HL10	HL12	HL14	SWISS20
ITALIC	HL8I	HL10I	HL12I	HL14I	SWISS20I
BOLD	HL8B	HL10B	HL12B	HL14B	SWISS20B
BOLD-ITALIC	HL8BI	HL10BI	HL12BI	HL14BI	SWISS20BI
BOLD-CONDENSED-CAPS	--	--	SWISS12B-CCAPS	--	--
CONDENSED-CAPS	--	--	SWISS12-CCAPS	--	--

Family DUTCH

	VERY-SMALL	SMALL	NORMAL	LARGE	VERY-LARGE
ROMAN	TR8	TR10	TR12	DUTCH14	DUTCH20
ITALIC	TR8I	TR10I	TR12I	DUTCH14I	DUTCH20I
BOLD	TR8B	TR10B	TR12B	DUTCH14B	DUTCH20B
BOLD-ITALIC	TR8BI	TR10BI	TR12BI	DUTCH14BI	DUTCH20BI

Family JESS

	NORMAL	LARGE
ROMAN	JESS13	JESS14
ITALIC	JESS13I	JESS14I
BOLD	JESS13B	JESS14B

Family EUREX

	HUGE
ITALIC	EUREX24I

11.4.5.4 Character Styles for LGP2 Fonts

Family FIX: Fonts are supported for all combinations of faces ROMAN, ITALIC, BOLD, and BOLD-ITALIC and sizes VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE.

Face	*LGP2 Font*
ROMAN	Courier

ITALIC	Courier-Oblique
BOLD	Courier-Bold
BOLD-ITALIC	Courier-Bold-Oblique

Size	*LGP2 Font Size*
VERY-SMALL	6 point
SMALL	7 point
NORMAL	9 point
LARGE	11 point
VERY-LARGE	14 point

Also, FIX.BOLD-EXTENDED.NORMAL maps to font Courier-Bold 9 point.

Family SWISS: Fonts are supported for all combinations of faces ROMAN, ITALIC, BOLD, and BOLD-ITALIC and sizes VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE.

Face	*LGP2 Font*
ROMAN	Helvetica
ITALIC	Helvetica-Oblique
BOLD	Helvetica-Bold
BOLD-ITALIC	Helvetica-Bold-Oblique

Size	*LGP2 Font Size*
VERY-SMALL	7 point
SMALL	8 point
NORMAL	10 point
LARGE	12 point
VERY-LARGE	16 point

Family DUTCH: Fonts are supported for all combinations of faces ROMAN, ITALIC, BOLD, and BOLD-ITALIC and sizes VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE.

Face	*LGP2 Font*
ROMAN	Times-Roman
ITALIC	Times-Oblique
BOLD	Times-Bold
BOLD-ITALIC	Times-Bold-Oblique

Size	*LGP2 Font Size*
VERY-SMALL	7 point
SMALL	8 point
NORMAL	10 point
LARGE	12 point
VERY-LARGE	16 point

Family Heading: Fonts are supported for all combinations of faces ROMAN, ITALIC, BOLD, and BOLD-ITALIC and sizes VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE.

Face	*LGP2 Font*
ROMAN	Times-Roman
ITALIC	Times-Oblique
BOLD	Times-Bold
BOLD-ITALIC	Times-Bold-Oblique

Size	*LGP2 Font Size*
VERY-SMALL	8 point
SMALL	9 point
NORMAL	12 point
LARGE	15 point
VERY-LARGE	19 point

11.5 Tables of Character Functions

11.5.1 Making a Character

make-character	Constructs a character, enabling you to set the bits and style.
set-char-bit	Changes a bit of a character and returns the new character.
code-char	Constructs a character given its code and bits fields.
make-char	Sets the bits field.

11.5.2 ASCII Characters

ascii-code	Returns the ASCII code for the argument.

char-to-ascii	Converts a character object with zero bits and style information to the corresponding ASCII code.
ascii-to-char	Converts an ASCII code to the corresponding character object.

11.5.3 Character Fields

For a diagram of the fields of a character: See the section "Fields of a Character", page 201.

The following functions can be used on characters:

char-code	Returns the value of the code field.
char-bits	Returns the value of the bits field.
char-font	Returns the value of the font field; since Genera characters have no font field this always returns zero.
sys:char-subindex	Returns the subindex field.
char-bit	Returns **t** if the specified bit is set.
si:char-style	Returns a character style object.

11.5.4 Character Predicates

The following predicates can be used on characters:

graphic-char-p	Checks whether the character is a printing character. Returns **t** if the character has no control bits set and is not a format effector.
upper-case-p	Returns **t** for uppercase characters.
lower-case-p	Returns **t** for lowercase characters.
both-case-p	Returns **t** for characters that exist in both cases.
alpha-char-p	Returns **t** for characters that are letters of the alphabet.
digit-char-p	Returns the "weight" of the digit character; for example, returns the integer 9 for the character #\9.
alphanumericp	Returns **t** for characters that are either letters or base-10 digits.
char-fat-p	Returns **t** if its argument is a fat character, otherwise **nil**.
characterp	Returns **t** for objects of type **character**.
standard-char-p	Returns **t** for objects of type **standard-char**.

string-char-p	Returns **t** for objects of type **string-char**.

For more information on the character types: See the section "Type Specifiers and Type Hierarchy for Characters", page 199.

11.5.5 Character Comparisons

None of the character comparisons ignore the character's bits.

11.5.5.1 Character Comparisons Affected by Case and Style

The following functions are used to compare characters exactly according to the code, case, character style, and bits fields.

char=	Returns **t** if the characters match.
char≠ or **char/=**	Returns **t** if the characters differ.
char<	Returns **t** if the first character is less than the second.
char>	Returns **t** if the second character is less than the first.
char≤ or **char<=**	Returns **t** if the first character is less than or equal to the second.
char≥ or **char>=**	Returns **t** if the second character is less than or equal to the first.

11.5.5.2 Character Comparisons Ignoring Case and Style

The following functions are used to compare characters according to the code and bits only, ignoring case and character style.

char-equal	Returns **t** if the characters match.
char-not-equal	Returns **t** if the characters differ.
char-lessp	Returns **t** if the first character is less than the second.
char-greaterp	Returns **t** if the second character is less than the first.
char-not-greaterp	Returns **t** if the first character is less than or equal to the second.
char-not-lessp	Returns **t** if the second character is less than or equal to the first.

11.5.6 Character Conversions

The following functions are used in changing the case of characters.

character	Coerces its argument to be a single character, if possible.
char-int	Converts a character to an integer.
int-char	Converts an integer to a character.
char-upcase	Returns the uppercase form of a character.
char-downcase	Returns the lowercase form of a character.
char-flipcase	Flips the case of a character.
digit-char	Returns the character that represents the specified "weight". For example, returns the character #\9 for the integer 9.

11.5.7 Character Names

char-name	Given a character object that has a name, returns the string that is the character's name.
name-char	Given a string that is the name of a character, returns the character object of that name.

11.5.8 Character Attribute Constants

The following constants represent the exclusive upper limits for the values of character attributes.

char-bits-limit	Upper limit for the value of the bits field.
char-code-limit	Upper limit for the value of the code field.
char-font-limit	Upper limit for the value in the font field; Genera characters do not have a font field so the value of **char-font-limit** is 1.

11.5.9 Character Bit Constants

char-control-bit	The control key bit; the value is 1.
char-hyper-bit	The hyper key bit; the value is 8.
char-meta-bit	The meta key bit; the value is 2.
char-super-bit	The super key bit; the value is 4.

12. Strings

12.1 Introduction to Strings

Strings are a type of one-dimensional array (a vector) whose elements are characters.

Symbolics Common Lisp supports 2 types of string arrays whose elements can be either of type **string-char**, or of type **character**. This is an extension of Common Lisp, where strings are arrays with elements restricted to type **string-char**.

The two types of Symbolics Common Lisp string arrays are also referred to as *thin* strings and *fat* strings.

Thin strings	These are arrays whose elements are standard characters of type **string-char**, with zero modifier bit and style fields. (In Zetalisp, this is the array type **sys:art-string.**)
Fat strings	These are arrays whose elements are wider characters of type **character** with bits holding information about modifier bits, style, and character code. (In Zetalisp, this is the array type **sys:art-fat-string.**) Some string operations ignore these extra bits.

See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. The way characters work, including multiple fonts and the extra bits from the keyboard, is explained in that section.

Examples:

```
(make-string 3 :initial-element #\s)  => "sss"    ; a thin string

(make-string 3 :initial-element #\s :element-type 'character)
=> "sss"   ; a fat string

(make-array 3 :element-type 'character :initial-element #\hyper-super-s)
 => "<H-S-S><H-S-S><H-S-S>"   ; a fat string
```

A further distinction between string types is based on array structure: the type **simple-array** holds a subtype of **string** called **simple-string**. This distinction is part of the Common Lisp standard, but is not very important for Symbolics machines.

string	An array, possibly with a fill pointer, whose contents are possibly shared with other array objects. Depending on the type of its elements, **string** can be thin or fat.
simple-string	A subtype of **string**: an array without a fill pointer, and whose contents are not shared with other array objects. Depending on the type of its elements, **simple-string** can be thin or fat.

See the section "Array Leaders", page 164. See the section "Displaced Arrays", page 166. Fill-pointers and the concept of shared arrays are discussed in these sections.

The string-specific function **make-string** creates a simple string that can be either thin or fat. The more general function **make-array** creates all types of strings.

Examples:

```
(make-string 3 :initial-element #\s)  => "sss"
(simple-string-p *) => T           ; a thin, simple, string

(string-fat-p (make-string 3 :initial-element #\super-s)) => T
                                   ; a fat, simple string

(make-array 3 :element-type 'character) => "..."
(simple-string-p *) => T           ; a fat, simple string

(make-array 3 :element-type 'string-char
              :initial-element #\$
              :fill-pointer 2) => "$$"

(stringp *) => T                   ; a thin, but not simple, string

(make-array 3 :element-type 'character  :fill-pointer 2) => ".."
(simple-string-p *) => NIL         ; a fat, but not simple, string
```

The printed representation of a string is its characters enclosed in quotation marks, for example, "**foo bar**". Strings are constants, that is, evaluating a string returns that string. Strings are the right data type to use for text processing.

Since strings are arrays, the usual array-referencing function **aref** is used to extract the characters of the string. For example:

```
(aref "frob" 1)  => #\r
```

Note that the character at the beginning of the string is element zero of the array (rather than one); as usual in Symbolics Common Lisp, everything is zero-based.

It is also valid to store into strings (using **setf** of **aref**). As with **rplaca** on lists, this changes the actual object; one must be careful to understand where side effects propagate to. When you are making strings that you intend to change later, you probably want to create an array with a fill-pointer so that you can change the length of the string as well as the contents. See the section "Array Leaders", page 164. The length of a string is always computed using **length**, so that if a string has a fill-pointer, its value is used as the length.

See also **intern**, which given a string returns "the" symbol with that print name. Similarly, **make-symbol** creates and returns an uninterned symbol with that print name.

12.2 How the Reader Recognizes Strings

The reader recognizes strings, written as a sequence of the characters contained in the string, and surrounded by the " (double quote) character.

Any double quote or escape character (for example, the \ backslash) in the sequence must be preceded by another \ escape character.

(Note: In Zetalisp, the / (slash) is the escape character, and it must be doubled when it occurs inside a string in Zetalisp code.)

Examples of strings:

```
"This is a typical string."
"That is known as a \"cons cell\" in Lisp."
```

Any | (vertical bar) inside a string need not be preceded by a backslash. Similarly, any double quote in the name of a symbol written using vertical-bar notation need not be preceded by a \.

The characters contained by the double quotes, taken from left to right, occupy locations with increasing indices within the string. The leftmost character is string element number 0, the next one is element number 1, and so on.

Note that the function **prinl** prints any string (not just a simple one) using this syntax, but the function **read** always constructs a simple string when it reads this syntax. The reader creates thin strings whenever it can.

12.3 Type Specifiers and Type Hierarchy for Strings

The type specifiers relating to strings are as follows:

character	string	array
string-char	simple-string	simple-array
standard-char		sequence
		vector

Details about each type specifier appear in its dictionary entry.

Figure 10 shows the relationships between the various data types relating to strings. For more on data types, type specifiers, and type-checking in Symbolics Common Lisp: See the section "Data Types and Type Specifiers", page 69.

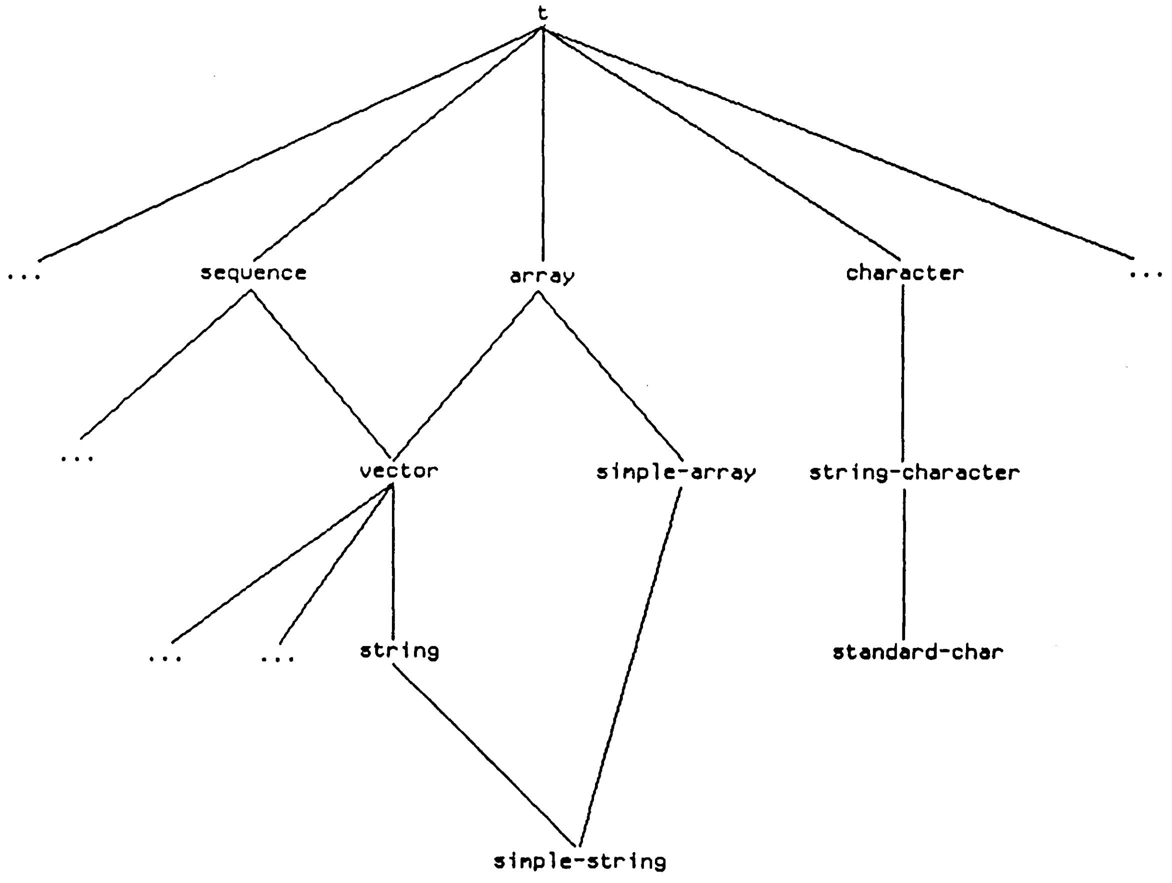


Figure 10. Symbolics Common Lisp String Data Types

A string is a specialized vector, or one-dimensional array, whose elements are of type **character**, or **string-char**. `string ≡ (or (vector string-char)(vector character))`. The type **string** can, therefore, be a subtype of the type **vector**.

Fat strings, that is strings of type `(vector character)`, are a Symbolics Common Lisp extension of Common Lisp.

The types `(vector string-char)` and `(vector character)` are *disjoint*; that is, a string cannot simultaneously be thin and fat.

The type **simple-string** is a subtype of the types **string**, and **simple-array**. **simple-string** is not, however, a subtype of the type **simple-vector**.

Any Lisp object can be tested to see whether it is a string whose elements are of type **character** or **string-char**, and whether it is a simple or a more complex string. See the section "String Type-Checking Predicates", page 229.

Some of the type specifier symbols for strings can be used also in type specifier lists for specialization and abbreviation of string data types. For example:

```
(typep (make-string 6 :initial-element #\b) '(simple-string 3)) => NIL
```

See the section "Type Specifier Lists", page 74.

12.4 Operations with Strings

There are several types of string operations, which can be done with specialized string functions or with more general-purpose sequence functions. The majority of these functions take any type of string argument. Note that whenever a sequence function must construct and return a new string, it always returns a simple string.

String-specific functions whose name begins with **string** accept a symbol instead of a string argument, provided that the operation never modifies that argument. The print name of the symbol is used. On the other hand, the more general sequence functions that can be applied to strings never accept symbols as sequences. (You can, however, apply the coercion function **string** to any argument to make it acceptable to a sequence function.)

The string-specific operations fall logically into ten major groups, as follows:

- String Type-checking Predicates
- String Access and Information
- String Construction
- String Conversion (case changes and pluralizing)
- String Manipulation (trimming and reversing)
- String Comparison Predicates (case-sensitive and case-insensitive comparisons)
- String Searches (case-sensitive and case-insensitive searches)
- ASCII strings
- String Input and Output
- Maclisp-Compatible String Functions

Generic counterparts of string-specific functions are listed in the summary tables of string functions. For more on sequence functions: See the section "Sequence Operations", page 188.

Several string functions, notably those involving string searches and comparison, are further subdivided into groups of functions that either respect or ignore string characteristics such as character style and alphabetic case.

See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

When strings have fill-pointers, string functions generally operate only on the active portion of the string (below the fill pointer).

String operations can be performed on specific portions of a string argument, and, where appropriate, in either direction. These user options are controlled by keyword arguments to the functions, as explained below. See the section "Keyword Arguments Delimit and Direct String Operations", page 228.

Many string functions have "destructive" as well as non-destructive versions. Functions beginning with the character "n" (for example, **string-nreverse**) destroy the original argument while operating on it; functions that do not have the "n" prefix (for example, **string-reverse**), return a modified *copy* of the original argument. Such pairs always appear together in the tables, with the non-destructive version listed first. Since it is highly undesirable to modify a string being used as the print name of a symbol, destructive functions cannot take symbols as arguments.

12.4.1 Case-Sensitive and Case-Insensitive String Comparisons

String comparisons compare every individual element of the string arrays; this is done by examining the various attributes of each character. The specific character attributes examined or ignored depend on whether the comparison is *case-sensitive* or *case-insensitive.*

case-sensitive comparison takes into account every single attribute of the characters compared, whereas *case-insensitive* comparison ignores the attributes specifying character *style* and character *case.*

Both case-sensitive and case-insensitive comparison functions compare attribute fields such as character code and modifier bits.

The string comparison and string searching functions call on character comparison functions, **char=** and **char-equal** for case-sensitive and case-insensitive comparison respectively.

Character objects and operations are explained elsewhere in detail. See the section "Characters", page 199. Here for convenience is a summary of some pertinent character attribute information.

Character style is a combination of three characteristics that describe how a character appears. These characteristics are the family, face, and size. The *family* field is a grouping that has typographic integrity, for example, Sans-Serif and Fix. The *face* field is a modification such as bold or italic. The *size* field is the size of the character.

Character case refers to the use of lower and upper case for alphabetic characters.

Character code identifies the particular character within its character set, in the same way that ASCII codes represent particular characters.

Modifier bits refers to the hyper, super, meta, and control keys on the keyboard.

The case-sensitive string comparison functions are distinguished by their use of algebraic comparison symbols as suffixes (for example, **string≠**, **string≥**); the case-insensitive string comparison functions have alphabetic symbols as suffixes (for example, **string-equal, string-not-equal, string-not-lessp**).

The case-sensitive string search functions often use the term **-exact** as part of their name; for example, **string-search-exact-char**; the case-insensitive string search functions omit this term, for example, **string-search-char**.

Here is an example of case-sensitive and case-insensitive character comparisons for various combinations of character style, character case, and modifier bits.

```
(let ((victims (list #\A #\a #\c-A
                (make-character #\A :style '(nil :italic nil)))))
  (loop for first-victims on victims
        for first-victim = (first first-victims)
        do
    (loop for second-victim in first-victims
          do
      (format T "~%~8<~c~;~> ~8<~c~;~> char= ~3s char-equal ~3s"
              first-victim
              second-victim
              (char= first-victim second-victim)
              (char-equal first-victim second-victim)))))
```

```
A        A        char= T   char-equal T
  ; case-sensitive comparision succeeds -- case matches

A        a        char= NIL char-equal T
  ; case sensitive comparision fails -- case$ differs

A        c-A      char= NIL char-equal NIL
  ; both comparisions fail -- modifier bit fields differ

A        A        char= NIL char-equal T
  ; case-sensitive comparision fails -- styles differ

a        a        char= T   char-equal T

a        c-A      char= NIL char-equal NIL
  ; both comparisions fail -- modifier bit fields differ

a        A        char= NIL char-equal T
  ; case sensitive comparision fails -- case and style differ

c-A      c-A      char= T   char-equal T
  ; both comparisons succeed -- modifier bit fields, case and style match

c-A      A        char= NIL char-equal NIL
  ; both comparisons fail -- modifier bit fields and style differ

A        A        char= T   char-equal T  NIL
  ; case -sensitive comparsion succeeds -- case and style match
```

12.4.2 Keyword Arguments Delimit and Direct String Operations

For the sake of efficiency, the majority of string-specific functions let you delimit the portion of the string to be operated on. Such functions have keyword arguments called **:start** and **:end**.

:start and **:end** must be non-negative integer indices into the string array. These keywords operate only on the "active" portion of the string, that is, the portion below the limit specified by the fill pointer, if there is one.

:start must be smaller than or equal to **:end**, else an error is signalled.

:start indicates the start position for the operation within the string. It defaults to zero (the start of the string).

:end indicates the position of the first element in the string *beyond* the end of the operation. It defaults to **nil** (the length of the string).

If both **:start** and **:end** are omitted, the entire string is processed by default.

If two strings are involved, the keyword arguments **:start1**, **:end1**, **:start2**, and **:end2** are used to specify substrings for each separate string argument.

For operations such as searches it can be useful to specify the direction in which the string is conceptually processed. This is controlled by the keyword argument **:from-end**.

Where this keyword is present, the function normally processes the string in the forward direction, but if a non-**nil** value is specified for **:from-end**, processing starts from the reverse direction. See the section "Case-Insensitive String Searches", page 240.

For some functions, the keyword **:count** is used to specify how many occurrences of an item should be affected. If **:count** is **nil**, or is not supplied, all matching items are affected.

12.4.3 String Type-Checking Predicates

These predicates test whether an object is a string of the recognized string types. The general type-checking predicate **typep** can also be used to test for strings. See the section "Determining the Type of an Object", page 79.

simple-string-p *object*	Returns **t** if *object* is a simple string array (one with no fill pointer and no displacement), and **nil** otherwise; accepts any object as an argument.
string-char-p *char*	Returns **t** if *char* can be stored into a thin string (that is, if it is a standard character; returns **nil** otherwise. Accepts character argument only.
string-fat-p *string*	Returns **t** if *string* is an array of fat characters, and **nil** otherwise. Accepts a string argument only.
stringp *object*	Returns **t** if *object* is either type of string, and **nil** otherwise; accepts any object as an argument.

12.4.4 String Access and Information

This group includes functions that access a string to extract a single character, or a specified number of characters (a substring) from it, or to return information, such as the length of the string.

In Symbolics Common Lisp the array-accessing function **aref** is useful for extracting a character from a string. If portability of your programs is a consideration, you might use the function **char** instead of **aref.**

The function **setf** can be used with **char** (or **aref**) to destructively replace a character within a string.

char *array* &rest *subscripts*	Accesses a single character element of a string. **char** and **aref** are equivalent in Symbolics Common Lisp.
schar *array* &rest *subscripts*	Same as **char**.
substring *string from* &optional *to area*	Extracts a substring from *string*. See also **subseq.**
nsubstring *string from* &optional *to area*	Extracts a substring from *string* but makes a displaced array instead of copying *string.*
string-length *string*	Returns the number of characters in *string.* See also **length.**
parse-integer *string* &key (*start* **0**) (*end* **nil**) (*radix* **10**) (*junk-allowed* **nil**) (*sign-allowed* **t**)	"Reads" a number from *string.* Returns **nil**, or a number found in *string*, plus the character position of the next unparsed character in *string.*
zl:parse-number *string* &optional *(from 0) to radix fail-if-not-whole-string*	"Reads" a number from *string.* Returns **nil**, or a number found in *string*, plus the character position of the next unparsed character in *string.* Use **parse-integer** instead.
sys:number-into-array *array n* &optional (*radix* **zl:base**) (*at-index* 0) (*min-columns* 0)	Deposits the printed representation of *number* into *array.*

12.4.5 String Construction

These string-specific functions coerce arguments to strings, construct simple, thin string arrays, or create strings by concatenation.

More complex character arrays can be constructed using the generic function **make-array.**

string, the first function listed in the table, works on strings, symbols, or characters, but does *not* work on lists or other sequences. The general function **coerce** converts a sequence of characters to a string, but does not accept symbols.

To get the string representation of a number or any other Lisp object, use **prinl-to-string**, **princ-to-string**, or **format**.

string *x*	Coerces *x* into a string. Returns a string, if *x* is a string, the print name of the symbol, if *x* is a symbol; if *x* is a character, a string containing that character is returned.
make-string *size* &key *initial-element element-type area*	Returns a simple string of thin or fat characters of length *size* initialized as specified by *initial-element* and created in the area specified by *area*. See also **make-sequence** and **make-array**.
string-append &rest *strings*	Returns a copy of its string arguments concatenated into a single string. See also **concatenate**.
string-nconc *modified-string* &rest *strings*	The destructive version of **string-append**.
string-nconc-portion *to-string* {*from-string from to*} ...	Adds information onto a string without consing intermediate substrings.
zl:string-nconc *to-string* &rest *strings*	Returns a concatenated string but modifies its first argument instead of copying it. Use **string-nconc** instead.

12.4.6 String Conversion

The specialized functions in this group pluralize a (sub)string, or change string case for the entire string, specified portions of the string, or for initial characters in words within the string. The keywords **:start** and **:end** delimit the portion of the string to be operated on. **:start** defaults to 0 (the entire string); **:end** defaults to **nil** (the length of the string). **:start** must be ≤ **:end**. Note that although only a portion of the string may be affected, the functions return a result of the same length as the entire string argument.

In the case of the functions that operate on words, a word is defined as a consecutive subsequence of alphanumeric characters or digits, delimited at both ends either by a non-alphanumeric character, or by the beginning or the end of the string.

Most of these functions have separate "destructive" versions, prefixed by the letter "n", for example, **nstring-upcase**.

The Zetalisp versions of these functions have optional starting and ending arguments. A further argument, *copy-p*, controls the function's effect on its argument: if *copy-p* is not **nil**, the function returns a *copy* of its argument; if *copy-p* is **nil**, the function alters the argument itself (it works destructively).

string-pluralize *string*	Returns a string containing the plural of the word in *string*.
string-upcase *string* &key *(start 0) (end* **nil)**	Returns a copy of string with lowercase characters capitalized.
nstring-upcase *string* &key (*start* 0) (*end* **nil)**	Returns *string* with lowercase characters capitalized.
string-downcase *string* &key *(start 0) (end* **nil)**	Returns a copy of string with uppercase characters replaced by lowercase.
nstring-downcase *string* &key (*start* 0) (*end* **nil)**	Returns *string* with uppercase characters replaced by lowercase.
string-capitalize *string* &key *(start 0) (end* **nil)**	Returns a copy of *string* with initial capitals for each case-modifiable word.
nstring-capitalize *string* &key (*start* 0) (*end* **nil)**	Returns *string* with initial capitals for each case-modifiable word.
string-capitalize-words *string* &key (*start* **0**) (*end* **nil)**	Returns a copy of *string* with initial capitals for each word, and with hyphens changed to spaces.
nstring-capitalize-words *string* &key (*start* **0**) (*end* **nil)**	Returns *string* with initial capitals for each word.
string-flipcase *string* &key *(start 0) (end* **nil)**	Returns a copy of *string* with uppercase characters replaced by lowercase, and vice versa.
nstring-flipcase *string* &key *(start 0) (end* **nil)**	Returns *string* with uppercase characters replaced by lowercase, and vice versa.
zl:string-pluralize *string*	Returns a string containing the plural of the word in *string*. Use **string-pluralize** instead.

zl:string-upcase *string* &optional (*start* **0**) *to* (*copy p* **t**)
Returns a copy of *string* with lowercase characters capitalized or the *string* itself is modified and returned. Use **string-upcase** and **nstring-upcase** instead.

zl:string-downcase *string* &optional (*start* **0**) *to* (*copy p* **t**)
Returns a copy of *string* with uppercase characters replaced by lowercase or the *string* itself is modified and returned. Use **string-downcase** and **nstring-downcase** instead.

zl:string-capitalize-words *string* &optional (*copy-p* **t**) *keep-hyphen*
Returns *string* with initial capitals for each word or the *string* itself is modified and returned. Use **string-capitalize-words** and **nstring-capitalize-words** instead.

zl:string-flipcase *string* &optional (*from* 0) *to* (*copy-p* **t**)
Returns a copy of *string* with uppercase characters replaced by lowercase and vice versa; returns modified *string*, if *copy-p* is **nil**. Use **string-flipcase** and **nstring-flipcase** instead.

12.4.7 String Manipulation

The functions in this group reverse the characters in a string, or trim off specified portions of a string; trimming can occur either at a specified location within the string or at either extremity.

Some string manipulation functions take the argument *char-set*. This argument can be represented as a list of characters, an array of characters, or a string of characters. In Zetalisp, *char-set* can be represented as a list of characters, or a string of characters.

Several sequence functions are available for replacing or removing specified string portions. See the section "Sequence Modification", page 194.

string-trim *char-set string*	Returns a *substring* of *string* with the characters in *char-set* stripped off the beginning and the end.
string-left-trim *char-set string*	Returns a *substring* of *string* with all characters in *char-set* stripped off the beginning.
string-right-trim *char-set string*	Returns a *substring* of *string* with all characters in *char-set* stripped off the end.

string-reverse *string* &key (*start* 0) (*end* **nil**)	Returns a copy of *string* with the order of characters reversed. See also **reverse**.
string-nreverse *string* &key (*start* 0) (*end* **nil**)	Returns *string* with the order of characters reversed. See also **nreverse**.
zl:string-trim *char-set string*	Returns a *substring* of *string* minus the characters in *char-set*. Use **string-trim** instead.
zl:string-left-trim *char-set string*	Returns a *substring* of *string* with all characters in *char-set* stripped off the beginning. Use **string-left-trim** instead.
zl:string-right-trim *char-set string*	Returns a *substring* of *string* with all characters in *char-set* stripped off the end. Use **string-right-trim** instead.
zl:string-reverse *string*	Returns a copy of *string* with the order of characters reversed. Use **string-reverse** instead.
zl:string-nreverse *string*	Returns *string* with the order of characters reversed. Use **string-nreverse** instead.

12.4.8 Case-Sensitive String Comparison Predicates

These predicates compare two strings or substrings of them, exactly, depending on all fields including character style, and alphabetic case. See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

The keywords (*:start1* **0**) and (*:start2* **0**) specify the character position (counting from 0) from which to *begin* the comparison; the keywords (*:end1* **nil**) and (*:end2* **nil**) specify the character position immediately *after* the end of the comparison. The start arguments default to **0** (compare strings in their entirety); the end arguments default to the length of the string. The start arguments must be ≤ the end arguments.

The predicates compare the strings in dictionary order. They return either the symbol **nil** or, generally, the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

For the Zetalisp versions of these predicates, the optional arguments, *idx1* and *idx2* specify the start point for the comparison, while *lim1* and *lim2* specify the character immediately after the end of the comparison. These Zetalisp predicates generally return either **t** or **nil**.

string= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if corresponding characters in the strings are identical or if the strings are unequal in length.

string≠ *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the characters in the two strings are not identical (same as **string/=**).

string/= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
A synonym of **string≠**.

string< *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char<**, or if *string1* is a proper substring of *string2.*

string≤ *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char≤**, or if *string1* is a substring of *string2* (same as **string<=**).

string<= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
A synonym of **string≤**.

string> *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char>**, or if *string2* is a proper substring of *string1.*

string≥ *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char≥**, or if *string2* is a substring of *string1* (same as **string>=**).

string>= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
A synonym of **string≥**.

string-exact-compare *string1 string2* &key (I[start1] **0**) (*start2* **0**) (*end1* **nil**) (*end2* **nil**)
Returns a positive number if *string1* > *string2,* zero if *string1* = *string2,* and a negative number if *string1* < *string2.*

sys:%string= *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison.

sys:%string-exact-compare *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison.

Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*.

zl:string= *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string=**, but returns **t** or **nil**.

zl:string≠ *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string≠**, but returns **t** or **nil**.

zl:string< *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string<**, but returns **t** or **nil**.

zl:string> *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string>**, but returns **t** or **nil**.

zl:string≤ *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string≤**, but returns **t** or **nil**.

zl:string≥ *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string≥**, but returns **t** or **nil**.

zl:string-exact-compare *string1 string2* &optional *idx1 idx2 lim1 lim2*
Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. Use **string-exact-compare** instead.

12.4.9 Case-Insensitive String Comparison Predicates

These predicates test strings ignoring character case and character style. See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

The keywords (*:start1* **0**) and (*:start2* **0**) specify the character position (counting from 0) from which to *begin* the comparison; the keywords (*:end1* **nil**) and (*:end2* **nil**) specify the character position immediately *after* the end of the comparison. The start arguments default to **0** (compare strings in their entirety); the end arguments default to the length of the string. The start arguments must be ≤ the end arguments.

The predicates compare the strings in dictionary order. They return either the symbol **nil** or, generally, the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

For the Zetalisp versions of these predicates, the optional arguments *idx1* and *idx2* specify the start point for the comparison, while *lim1* and *lim2* specify the character immediately after the end of the comparison. These Zetalisp predicates generally return either **t** or **nil**.

string-equal *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string=**.

string-not-equal *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string/=**.

string-lessp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string<**.

string-greaterp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string>**.

string-not-greaterp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string<=**.

string-not-lessp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string>=**.

string-compare *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string-exact-compare**.

sys:%string-equal *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison. Case-insensitive version of **sys:%string=**.

sys:%string-compare *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison. Case-insensitive version of **sys:%string-exact-compare**.

zl:string-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string=**. Use **string-equal** instead.

zl:string-not-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string≠**. Like **string-not-equal** but returns **t** or **nil**.

zl:string-lessp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string<**. Like **string-lessp** but returns **t** or **nil**.

zl:string-greaterp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string>**. Like **string-greaterp** but returns **t** or **nil**.

zl:string-not-lessp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string-not-lessp** but returns **t** or **nil**.

zl:string-not-greaterp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string-not-greaterp** but returns **t** or **nil**.

zl:string-compare *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string-exact-compare**. Use **string-compare** instead.

12.4.10 Case-Sensitive String Searches

The following string-specific functions search a string argument, looking for the presence (or absence) of a specified character (*char*), or of a string (*key*). The functions use **char=**, and, as denoted by the fact that they all contain the word **exact** as part of their name, the functions compare all fields of the character, including character style and alphabetic case. See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

The keywords **:start1** and **:start2** specify the character position (counting from 0) from which to *begin* the comparison; the keywords **:end1** and **:end2** specify the character position immediately *after* the end of the comparison. The start arguments default to **0** (compare strings in their entirety); the end arguments default to the length of the string. The start arguments must be ≤ the end arguments.

The functions return either **nil** (unsuccessful), or the position of the first successful occurrence of the item sought for. Typically, this is the position of the *leftmost* occurrence. If the keyword argument **:from-end** is present and has a non-**nil** value, the function returns the position of the *rightmost* element satisfying the test, as though the search direction had been reversed. Position is always counted from the beginning of the string.

Generic sequence functions can also be used to locate one or more string elements satisfying some test. See the section "Searching for Sequence Items", page 196.

The case sensitive search functions have parallel versions that work in case-insensitive fashion. For a comparison of the case-sensitive and case-insensitive versions: See the section "Summary of String Searching Functions", page 243.

For the Zetalisp versions of case-sensitive search functions the optional arguments *from* and *to* let you specify the character position, (counting from 0) from which to begin and end the search, respectively. The optional arguments *key-start* and *key-end* let you specify substrings of the string searched for.

from defaults to 0 (search the entire string), while *to* defaults to **nil** (the length of the string). *key-start* and *key-end* default in analogous fashion.

In Zetalisp there are separate reverse-search functions. These return a string position from the *beginning* of the string, even though the search begins at the end of the string.

string-search-exact-char *char string* &key *from-end* (*start* **0**) (*end* **nil**)
Searches *string* or a specified substring for *char*. Returns **nil**, or the position of the first occurrence of *char*.

string-search-not-exact-char *char string* &key *from-end* (*start* **0**) (*end* **nil**)
Searches *string* or a specified substring for occurrences of any character other than *char*. Returns **nil**, or the position of the first character that does not match *char*.

string-search-exact *key string* &key *from-end* (*start1* **0**) (*end1* **nil**) (*start2* **0**) (*end2* **nil**)
Searches for *key* in *string*. Substrings of either argument can be specified. Returns **nil**, or the position of the first character of *key* occurring in *string*.

zl:string-search-exact-char *char string* &optional (*from* **0**) *to*
Searches *string* or a specified substring for *char*. Returns **nil**, or the position of the first occurrence of *char*. Use equivalent function **string-search-exact-char** instead.

zl:string-search-not-exact-char *char string* &optional (*from* **0**) *to*
Searches *string* or a specified substring for occurrences of any character other than *char*. Returns either **nil**, or the position of the first character that does not match *char*. Use equivalent function **string-search-not-exact-char** instead.

zl:string-reverse-search-exact-char *char string* &optional *from* (*to* **0**)
Searches *string* or a specified substring, starting from the end of the string. Returns **nil**, or the position of the last occurrence of *char*.

zl:string-reverse-search-not-exact-char *char string* &optional *from* (*to* **0**)
Searches *string*, or a specified substring for occurrences of any character other than *char*, starting from the end of the string. Returns **nil**, or the position of the last occurrence of a character other than *char*.

zl:string-search-exact *key string* &optional (*from* **0**) *to* (*key-start* **0**) *key-end*
Searches for *key* in *string*. Substrings of either

argument can be specified. Returns **nil**, or the position of the first character of *key* occurring in *string*. Use equivalent function **string-search-exact** instead.

zl:string-reverse-search-exact *key string* &optional (*from* **0**) *to* (*key-start* **0**) *key-end*
Searches for *key* in *string*, starting from the end of *string*. Substrings of either argument can be specified. Returns **nil**, or the position of the last occurrence of the first character of *key* in *string*.

sys:%string-search-exact-char *char string start end*
A low-level search, possibly more efficient than other searching functions.

12.4.11 Case-Insensitive String Searches

These functions search a string argument, looking for the presence or absence of a specified character (*char*), string *key*, or a character that is part of a character set (*char-set*). The functions use **char-equal**, which ignores character style and alphabetic case. See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

The keywords **:start1** and **:start2** specify the character position (counting from 0) from which to *begin* the comparison; the keywords **:end1** and **:end2** specify the character position immediately *after* the end of the comparison. The start arguments default to **0** (compare strings in their entirety); the end arguments default to the length of the string. The start arguments must be ≤ the end arguments.

Several functions take the argument *char-set*. This argument can be represented as a list of characters, an array of characters, or a string of characters.

The functions return either **nil** (unsuccessful), or the position of the first successful occurrence of the item sought for. Typically, this is the position of the *leftmost* occurrence. If the keyword argument **:from-end** is present and has a non-**nil** value, the function returns the position of the *rightmost* element satisfying the test, as though the search direction had been reversed. Position is always counted from the beginning of the string.

Except for the functions that search for *char-set*, the case insensitive search functions have parallel versions that work in case-sensitive fashion. For a comparison of the case-sensitive and case-insensitive versions: See the section "Summary of String Searching Functions", page 243.

General sequence functions can also be used to locate one or more string elements satisfying some test. See the section "Searching for Sequence Items", page 196.

Note that the sequence functions use **eql** to perform the test. We recommend that in specifying the **:test** keyword argument you use a specific comparison function such as **char-equal**.

For the Zetalisp versions of case-insensitive string searching functions the optional arguments *from* and *to* let you specify the character position, (counting from 0) from which to begin and end the search, respectively. The optional arguments *key-start* and *key-end* let you specify substrings of the string searched for.

from defaults to 0 (search the entire string), while *to* defaults to **nil** (the length of the string). *key-start* and *key-end* default in analogous fashion.

The argument *char-set* can be represented in Zetalisp as a list of characters or a string of characters.

In Zetalisp there are separate reverse-search functions. These return a string position from the *beginning* of the string, even though the search begins at the end of the string.

string-search-char *char string* &key *from-end* (*start* **0**) (*end* **nil**)
Searches *string* or a specified substring for *char*. Returns **nil**, or the position of the first occurrence of *char*.

string-search-not-char *char string* &key *from-end* (*start* **0**) (*end* **nil**)
Searches *string* or a specified substring for occurrences of any character other than *char*. Returns either **nil**, or the position of the first character that does not match *char*.

string-search *key string* &key *from-end* (*start1* **0**) (*end1* **nil**) (*start2* **0**) (*end2* **nil**)
Searches for the string *key* in *string*. Substrings of either argument can be specified. Returns **nil**, or the position of the first character of *key* occurring in *string*.

string-search-set *char-set string* &key *from-end* (*start* **0**) (*end* **nil**)
Searches *string*, or a specified substring, for a character that is in *char-set*. Returns **nil**, or the position of the first character that is **char-equal** to some element of *char-set*.

string-search-not-set *char-set string* &key *from-end* (*start* **0**) (*end* **nil**)
Searches *string*, or a specified substring, for occurrences of any character that is not in *char-set*. Returns **nil**, or the position of the first character that is not **char-equal** to some element of *char-set*.

zl:string-search-char *char string* &optional (*from* **0**) *to*
Searches *string* or a specified substring for *char*.

Returns **nil**, or the position of the first occurrence of *char*. Use equivalent function **string-search-char** instead.

zl:string-search-not-char *char string* &optional (*from* **0**) *to*

Searches *string* or a specified substring for occurrences of any character other than *char*. Returns either **nil**, or the position of the first character that does not match *char*. Use equivalent function **string-search-not-char** instead.

zl:string-reverse-search-char *char string* &optional *from* (*to* **0**)

Searches *string* or a specified substring, starting from the end of the string. Returns **nil**, or the position of the first occurrence of *char* in *string*.

zl:string-reverse-search-not-char *char string* &optional *from* (*to* **0**)

Searches *string*, or a specified substring for occurrences of any character other than *char*, starting from the end of the string. Returns **nil**, or the position of the first occurrence of a character other than *char*.

zl:string-search *key string* &optional (*from* **0**) *to* (*key-start* **0**) *key-end*

Searches for *key* in *string*. Substrings of either argument can be specified. Returns **nil**, or the position of the first character of *key* occurring in *string*. Use equivalent function **string-search** instead.

zl:string-reverse-search *key string* &optional from (*to* **0**) (*key-start* **0**) *key-end*

Searches for *key* in *string*, starting from the end of *string*. Substrings of either argument can be specified. Returns **nil**, or the position of the first occurrence of the first character of *key* in *string*.

sys:%string-search-char *char string start end*

A low-level, possibly more efficient search.

zl:string-search-set *char-set string* &key *from-end* (*start* **0**) (*end* **nil**)

Searches through *string*, or a specified substring, for a character that is in *char-set*. Returns **nil**, or the position of the first character that is **char-equal** to some element of *char-set*. Use equivalent function **string-search-set** instead.

zl:string-search-not-set *char-set string* &key *from-end* (*start* **0**) (*end* **nil**)

Searches *string*, or a specified substring, for occurrences of any character that is not in *char-set*. Returns **nil**, or the position of the first character that is not **char-equal** to some element of *char-set*. Use equivalent function **string-search-not-set** instead.

zl:string-reverse-search-set *char-set string* &optional *from* (*to* **0**)
Searches through *string*, or a specified substring in reverse order, looking for a character that is in *char-set*. Returns **nil**, or the position of the first character that is **char-equal** to some element of *char-set*.

zl:string-reverse-search-not-set *char-set string* &optional *from* (*to* **0**)
Searches through *string*, or a specified substring in reverse order, looking for a character that is not in *char-set*. Returns **nil**, or the position of the first character that is not **char-equal** to some element of *char-set*.

12.4.12 Summary of String Searching Functions

Case-sensitive Version	*Case-insensitive Version*
string-search-exact-char **zl:string-search-exact-char**	**string-search-char** **zl:string-search-char**
string-search-not-exact-char **zl:string-search-not-exact-char**	**string-search-not-char** **zl:string-search-not-char**
zl:string-reverse-search-exact-char	**zl:string-reverse-search-char**
zl:string-reverse-search-not-exact-char	**zl:string-reverse-search-not-char**
string-search-exact **zl:string-search-exact**	**string-search** **zl:string-search**
zl:string-reverse-search-exact	**zl:string-reverse-search**

sys:%string-search-exact-char	**sys:%string-search-char**
[None]	**string-search-set** **zl:string-search-set**
[None]	**string-search-not-set** **zl:string-search-not-set**
[None]	**zl:string-reverse-search-set**
[None]	**zl:string-reverse-search-not-set**

12.4.13 ASCII String Functions

string-to-ascii *lispm-string*	Converts *lispm-string* to a **sys:art-8b** array containing ASCII character codes.
ascii-to-string *ascii-array*	Converts *ascii-array*, a **sys:art-8b** array representing ASCII characters, into a Lisp string.

See the section "ASCII Characters", page 217.

12.4.14 String Input and Output

The generic functions **read** and **write** can be used in conjunction with several specialized functions that operate on character streams. These functions let you create I/O streams that input from or output to a string rather than to a real I/O device.

The two tables below summarize string input and output functions, and the two variables that control the printing of strings. String I/O is fully documented in *Reference Guide to Streams, Files, and I/O.*

12.4.14.1 String Input and Output Functions

write-string *string* &optional *output-stream* &key (*start* **0**) *end*

Writes the characters of the specified substring of *string* to *output-stream.* Returns the *string*, not the substring. See the section "Output Functions" in *Reference Guide to Streams, Files, and I/O.*

with-input-from-string (*stream string* &key *index* (start **0**) *end)* &body *body*

Executes *body* with *stream* bound to a character input stream that supplies successive characters from the value of *string*. See the special form **with-open-file** in *Reference Guide to Streams, Files, and I/O.*

with-output-to-string (*stream* &optional *string* &key *index*) &body *body*
Executes *body* as an implicit **progn** with *stream* bound to a character output stream; all output to that stream is saved in a string. See the special form **with-open-file** in *Reference Guide to Streams, Files, and I/O.*

make-string-input-stream *string* &optional (*start* **0**) *end*
Returns an input stream that supplies the characters in the substring of *string* delimited by *start* and *end*. See the section "Stream Operations" in *Reference Guide to Streams, Files, and I/O.*

make-string-output-stream
Returns an output stream that accumulates output for **get-output-stream-string**. See the section "Stream Operations" in *Reference Guide to Streams, Files, and I/O.*

get-output-stream-string *stream*
Using a stream produced by **make-string-output-stream**, returns a string containing all characters output to the stream so far. See the section "Stream Operations" in *Reference Guide to Streams, Files, and I/O.*

write-to-string *object* &key *escape radix base circle pretty level length case gensym array readably array-length string-length pretty level length case gensym array structure-contents*
Returns the object, printed as if by **write**, with the characters that would be output made into a string. The keywords specify values for controlling the printed representation; each defaults to the value of the corresponding global variable. See the section "Output Functions" in *Reference Guide to Streams, Files, and I/O.*

read-from-string *string* &optional (*eof-errorp* **t**) *eof-value* &key (*start* **0**) *end preserve-whitespace*
The characters of *string* are given successively to the Lisp reader, and the Lisp object built by the reader is returned. Returns the object read,

and the index of the first character in the string not read (if entire string was read, returns the length of the string). See the section "Input Functions" in *Reference Guide to Streams, Files, and I/O.*

zl:with-input-from-string (var string &optional index limit) &body body
Evaluates the forms in *body* with *var* bound to a stream that reads characters from the string *string*. See the special form **with-open-file** in *Reference Guide to Streams, Files, and I/O.*

zl:with-output-to-string (var &optional (string **nil** string-p) index) &body body
Provides a variety of ways to send output to a string through an I/O stream. See the special form **with-open-file** in *Reference Guide to Streams, Files, and I/O.*

zl:read-from-string *string* &optional (*eof-option* **'si:no-eof-option**) (*start* **0**) *end*
The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned. See the section "Input Functions" in *Reference Guide to Streams, Files, and I/O.*

12.4.14.2 Control Variables for Printing Strings

print-case Controls the case (upper, lower, or mixed) in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used. See the section "Output Functions" in *Reference Guide to Streams, Files, and I/O.*

print-string-length Controls the number of string characters to print. See the section "Output Functions" in *Reference Guide to Streams, Files, and I/O.*

12.4.15 Maclisp-Compatible String Functions

The following functions are provided primarily for compatibility with older versions of Lisp, notably Maclisp.

alphalessp *string1 string2* Used with strings, is equivalent to the case-insensitive string comparison function, **string-lessp**; also accepts numbers, lists, and other objects.

zl:getchar *string index*	Returns the *index*th character of *string* (counting from 1) as a symbol. Use **aref** instead.
zl:getcharn *string index*	Returns the *index*th character of *string* (counting from 1) as a character. Use **aref** instead.
zl:ascii *x*	Returns a symbol whose printname is the character rather than an integer; the symbol is interned in the current package.
zl:maknam *char-list*	Returns an uninterned symbol whose print-name is a string made up of the characters in *char-list*.
zl:implode *char-list*	Returns a symbol whose print-name is a string made up of the characters in *char-list*; the symbol is interned in the current package.

PART III.

Defining Function Usage

13. Functions and Dynamic Closures

13.1 Functions

13.1.1 What Is a Function?

Functions are the basic building blocks of Lisp programs. There are many different kinds of functions in Symbolics-Lisp. Here are the printed representations of examples of some of them:

```
foo
(lambda (x) (car (last x)))
(si:digested-lambda (lambda (x) (car (last x)))
                    (foo) 2049 262401 nil (x) nil (car (last x)))
#<dtp-compiled-function append 1424771>
#<lexical-closure (lambda ** **) 7371705>
#<lexical-closure (:internal foo 0) 7372462>
#<dtp-closure 1477464>
```

These all have one thing in common: a function is a Lisp object that can be applied to arguments. All of the above objects can be applied to some arguments and will return a value. Functions are Lisp objects and so can be manipulated in all the usual ways: you can pass them as arguments, return them as values, and make other Lisp objects refer to them. See the function **functionp** in *Symbolics Common Lisp: Language Dictionary.*

13.1.2 Function Specs

The name of a function does not have to be a symbol. Various kinds of lists describe other places where a function can be found. A Lisp object that describes a place to find a function is called a *function spec.* ("Spec" is short for "specification".) Here are the printed representations of some typical function specs:

```
foo
(:property foo bar)
(flavor:method speed ship)
(:internal foo 1)
(:within foo bar)
(:location #<dtp-locative 7435216>)
```

Function specs have two purposes: they specify a place to remember a function, and they serve to *name* functions. The most common kind of function spec is a symbol, which specifies that the function cell of the symbol is the place to remember the function. Function specs are not the same thing as functions. You cannot, in general, apply a function spec to arguments. The time to use a function spec is when you want to do something to the function, such as define it, look at its definition, or compile it.

Some kinds of functions remember their own names, and some do not. The "name" remembered by a function can be any kind of function spec, although it is usually a symbol. (See the section "What is a Function?", page 251.) In that section, the example starting with the symbol **si:digested-lambda** and the one whose printed representation includes **sys:dtp-compiled-function**, remember names (the function specs **foo** and **append** respectively). The others do not remember their names, except that the ones starting with **sys:lexical-closure** and **sys:dtp-closure** might contain functions that do remember their names. The second **sys:lexical-closure** example contains the function whose name is **(:internal** foo 0).

To *define a function spec* means to make that function spec remember a given function. This is done with the **fdefine** function; you give **fdefine** a function spec and a function, and **fdefine** remembers the function in the place specified by the function spec. The function associated with a function spec is called the *definition* of the function spec. A single function can be the definition of more than one function spec at the same time, or of no function specs.

To *define a function* means to create a new function, and define a given function spec as that new function. This is what the **defun** special form does. Several other special forms such as **defmethod** and **defselect** do this too.

These special forms that define functions usually take a function spec, create a function whose name is that function spec, and then define that function spec to be the newly created function. Most function definitions are done this way, and so usually if you go to a function spec and see what function is there, the function's name is the same as the function spec. However, if you define a function named **foo** with **defun**, and then define the symbol **bar** to be this same function, the name of the function is unaffected; both **foo** and **bar** are defined to be the same function, and the name of that function is **foo**, not **bar**.

A function spec's definition in general consists of a *basic definition* surrounded by *encapsulations*. Both the basic definition and the encapsulations are functions, but of recognizably different kinds. What **defun** creates is a basic definition, and usually that is all there is. Encapsulations are made by function-altering functions such as **trace** and **advise**. When the function is called, the entire definition, which includes the tracing and advice, is used. If the function is "redefined" with **defun**, only the basic definition is changed; the encapsulations are left in place. See the section "Encapsulations", page 263.

A function spec is a Lisp object of one of the following types:

a symbol

The function is remembered in the function cell of the symbol. See the section "The Function Cell of a Symbol", page 125. Function cells and the primitive functions to manipulate them are explained in that section.

(**:property** *symbol property*)

The function is remembered on the property list of the symbol; doing (**zl:get** *symbol property*) would return the function. 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).

(**flavor:method** *generic-function flavor-name options...*)

This function spec names the method implemented for *generic-function* on instances of *flavor-name.* (*generic-function* can be the name of a generic function or a message.) The function is remembered inside internal data structures of the flavor system.

(**:handler** *generic-function flavor-name*)

This is a name for the function actually called when *generic-function* is called on an instance of the flavor *flavor-name.* (*generic-function* can be the name of a generic function or a message.) A handler is different than a method in the following way: you define one or more methods in source files, but it is the flavor system that consults all the available methods and constructs a handler from them. In the simplest case, the handler is the method written to perform *generic-function* on instances of *flavor-name.* In other cases, the handler might be a method inherited from a component flavor, or a *combined method* that includes several methods combined in a manner prescribed by the type of method combination. 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** and **advise**.

(**flavor:wrapper** *generic-function flavor*)

This function spec names a wrapper. If you trace a wrapper, note that wrappers are executed at compile time, being macros.

(**flavor:whopper** *generic-function flavor*)

This function spec names a whopper.

(**:location** *pointer*)

The function is stored in the cdr of *pointer*, which can be a locative or a list. This is for pointing at an arbitrary place which there is no other way to describe. This form of function spec is not useful in **defun** (and related special forms) because the reader has no printed representation for locative pointers and always creates new lists; these function specs are intended for programs that manipulate functions. See the section "How Programs Manipulate Definitions", page 261.

(**:within** *within-function function-to-affect*)

This refers to the meaning of the symbol *function-to-affect*, but only where it occurs in the text of the definition of *within-function*. If you define this function spec as anything but the symbol *function-to-affect* itself, then that symbol is replaced throughout the definition of *within-function* by a new symbol which is then defined as you specify. See the section "Encapsulations", page 263.

(**:internal** *function-spec number*)

Some Lisp functions contain internal functions, created by **(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. *function-spec* is the containing function. *number* is a sequence number; the first internal function the compiler comes across in a given function is numbered 0, the next 1, and so on.

(**:internal** *function-spec number name*)

Some Lisp functions contain internal functions, created by **flet** or **labels** forms. *function-spec* is the containing function. *number* is a sequence number; the first internal function the compiler comes across in a given function is numbered 0, the next 1, and so on. *name* is the name of the internal function.

Here is an example of the use of 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 puts a function on **foo**'s **bar-maker** property. Now you can say:

```
(funcall (get 'foo 'bar-maker) 'baz)
```

Unlike the other kinds of function spec, a symbol *can* be used as a function. If you apply a symbol to arguments, the symbol's function definition is used instead. If the definition of the first symbol is another symbol, the definition of the second symbol is used, and so on, any number of times. But this is an exception; in general, you cannot apply function specs to arguments.

A keyword symbol that identifies function specs (can appear in the car of a list that is a function spec) is identified by a **sys:function-spec-handler** property whose value is a function which implements the various manipulations on function specs of that type. The interface to this function is internal and not documented here.

For compatibility with Maclisp, the function-defining special forms **defun**, **macro**, and **defselect** (and other defining forms built out of them, such as **zl:defunp** and **defmacro**) also accept a list:

```
(symbol property)
```

as a function name. This is translated into:

```
(:property symbol property)
```

symbol must not be one of the keyword symbols which identifies a function spec, since that would be ambiguous.

13.1.3 Simple Function Definitions

See the section "Function-Defining Special Forms", page 258. Information on defining functions, and other ways of doing so, are discussed in that section.

13.1.4 Operations the User Can Perform on Functions

Here is a list of the various things a user (as opposed to a program) is likely to want to do to a function. In all cases, you specify a function spec to say where to find the function.

To print out the definition of the function spec with indentation to make it legible, use **grindef**. This works only for interpreted functions. If the definition is a compiled function, it cannot be printed out as Lisp code, but its compiled code can be printed by the **disassemble** function.

To find out about how to call the function, you can ask to see its documentation, or its argument names. (The argument names are usually chosen to have mnemonic significance for the caller). Use **arglist** to see the argument names and **documentation** to see the documentation string. There are also editor commands for doing these things: the `c-sh-D` and `m-sh-D` commands are for looking at a function's documentation, and `c-sh-A` is for looking at an argument list. `c-sh-A` does not ask for the function name; it acts on the function that is called by the innermost expression that the cursor is inside. Usually this is the function that is called by the form you are in the process of writing.

You can also find out about the function using **describe-function**. It shows the arglist, values, and any Common Lisp proclaims for a function spec.

describe-function *fspec* &key (*stream* ***standard-output***) *Function*

> **describe-function** shows the arglist, values and proclaims for *fspec*. The **:stream** argument enables you to output the description to any stream.

```
(describe-function 'locativep) =>
Debugging info:
  ARGLIST (OBJECT)
  SYS:FUNCTION-PARENT (LOCATIVEP DEFINE-TYPE-PREDICATE)
Proclaimed properties:
  NOTINLINE
NIL
```

You can see the function's debugging info alist by means of the function **debugging-info**.

When you are debugging, you can use **trace** to obtain a printout or a break loop whenever the function is called. You can customize the definition of the function, either temporarily or permanently, using **advise**.

13.1.5 Kinds of Functions

There are many kinds of functions in Symbolics-Lisp. This section briefly describes each kind of function. Note that a function is also a piece of data and can be passed as an argument, returned, put in a list, and so forth.

Before we start classifying the functions, we will first discuss something about how the evaluator works. 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 definition of the symbol is a function, then the function is just applied to the result of evaluating the rest of the subforms. If the definition is a list whose car is **special**, then it is either a macro form or a special form. For more information about macro forms: See the section "Macros", page 285. What about special forms?

Conceptually, the evaluator knows specially about all special forms (hence their name). However, the Symbolics-Lisp implementation actually uses the definition of symbols that name special forms as places to hold pieces of the evaluator. The definitions of such symbols as **prog**, **do**, **and**, and **or** actually hold Lisp objects, which we call *special functions*. Each of these functions is the part of the Lisp interpreter that knows how to deal with that special form. Normally you do not have to know about this; it is just part of how the evaluator works.

Many of the special forms in Zetalisp are implemented as macros. They are implemented this way because it is easier to write a macro than to write both a new part of the interpreter (a special function) and a new *ad hoc* module in the compiler. However, they are sometimes documented as special forms, rather than macros, because you should not in any way depend on the way they are implemented.

There are four kinds of functions, classified by how they work.

1. *Interpreted* functions, which are defined with **defun**, represented as list structure, and interpreted by the Lisp evaluator.

2. *Compiled* functions, which are defined by **compile** or by loading a bin file, are represented by a special Lisp data type, and are executed directly by the machine.

3. Various types of Lisp objects that can be applied to arguments, but when they are applied they dig up another function somewhere and apply it instead. These include symbols, dynamic and lexical closures, and instances.

4. Various types of Lisp objects that, when used as functions, do something special related to the specific data type. These include arrays and stack groups.

13.1.5.1 Interpreted Functions

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, an interpreted function can be printed out and read back in (it has a printed representation that the reader understands), and it can be pretty-printed. See the section "Formatting Lisp Code" in *Reference Guide to Streams, Files, and I/O*. It can also be opened up and examined with the usual functions for list-structure manipulation.

There are two kinds of interpreted functions: **lambdas** and **si:digested-lambdas**. A **lambda** function is the simplest kind. It is a list that looks like this:

(lambda *lambda-list form1 form2*...)

The symbol **lambda** identifies this list as a **lambda** function. *lambda-list* is a description of what arguments the function takes. See the section "Evaluating a Function Form", page 504. The *forms* make up the body of the function. When the function is called, the argument variables are bound to the values of the arguments as described by *lambda-list*, and then the forms in the body are evaluated, one by one. The value of the function is the value of its last form.

An **si:digested-lambda** is like a **lambda**, but contains extra elements in which the system remembers the function's name, its documentation, a preprocessed form of its lambda-list, and other information. Having the function's name there allows the Debugger and other tools to give the user more information. This is the kind of function that **defun** creates. The interpreter turns any lambdas it is asked to apply into digested-lambdas, using **rplaca** and **rplacd** to modify the list structure of the original lambda-expression.

13.1.5.2 Compiled Functions

The Lisp function compiler converts **lambda** functions into compiled functions. A compiled function's printed representation looks like:

```
    #<dtp-compiled-function append 1424771>
```

The object contains machine code that does the computation expressed by the function; it also contains a description of the arguments accepted, any constants required, the name, documentation, and other things. Unlike Maclisp "subr-objects", compiled functions are full-fledged objects and can be passed as arguments, stored in data structure, and applied to arguments.

13.1.5.3 Other Kinds of Functions

A dynamic closure is a kind of function that contains another function and a set of special variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When that returns, the closure bindings are removed. Dynamic closures are created by the **zl:closure** function and the **zl:let-closed** special form. See the section "Dynamic Closures", page 265.

A lexical closure is a kind of function that contains another function and a set of local variable bindings. A lexical closure is created by reference to an internal functions. Invocation of a lexical closure simply provides the necessary data linkage for a function to run in the environment in which the closure was made. See the section "Lexical Scoping", page 517.

An instance is a message-receiving object that has some state and a table of message-handling functions (called *methods*). See the section "Flavors", page 353.

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 works this way 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 the section "Stack Groups" in *Internals, Processes, and Storage Management.*

13.1.6 Function-Defining Special Forms

defun is a special form that is put in a program to define a function. **defsubst** and **macro** are others. This section explains how these special forms work, how they relate to the different kinds of functions, and how they connect to the rest of the function-manipulation system.

Function-defining special forms typically take as arguments a function spec and a description of the function to be made, usually in the form of a list of argument names and some forms that constitute the body of the function. They construct a function, give it the function spec as its name, and define the function spec to be the new function. Different special forms make different kinds of functions.

defun and **defsubst** both make an **si:digested-lambda** function. **macro** makes a macro; though the macro definition is not really a function, it is like a function as far as definition handling is concerned.

These special forms are used in writing programs because the function names and bodies are constants. Programs that define functions usually want to compute the functions and their names, so they use **fdefine**.

All of these function-defining special forms alter only the basic definition of the function spec. Encapsulations are preserved. See the section "Encapsulations", page 263.

The special forms only create interpreted functions. There is no special way of defining a compiled function. Compiled functions are made by compiling interpreted ones. The same special form that defines the interpreted function, when processed by the compiler, yields the compiled function.

Note that the editor understands these and other "defining" special forms (for example, **defmethod**, **defvar**, **defmacro**, and **zl:defstruct**) to some extent, so that when you ask for the definition of something, the editor can find it in its source file and show it to you. The general convention is that anything that is used at top level (not inside a function) and starts with **def** should be a special form for defining things and should be understood by the editor. **defprop** is an exception.

defun

The **defun** special form (and the **zl:defunp** macro that expands into a **defun**) are used for creating ordinary interpreted functions. See the section "Simple Function Definitions", page 255.

For Maclisp compatibility, a *type* symbol can be inserted between *name* and *lambda-list* in the **defun** form. The following types are understood:

zl:expr	The same as no type.
zl:fexpr	Defines a special form that operates like a Maclisp fexpr. The special form can only be used in interpreted functions and in forms evaluated at top-level, since the compiler has not been told how to compile it.
macro	A macro is defined instead of a normal function.

If *lambda-list* is a non-**nil** symbol instead of a list, the function is recognized as a Maclisp *lexpr* and it is converted in such a way that the **zl:arg**, **zl:setarg**, and **zl:listify** functions can be used to access its arguments.

defsubst	The **defsubst** special form is used to create inline functions. It is used just like **defun** but produces a function that acts normally when applied, but can also be open-coded (incorporated into its callers) by the compiler. See the section "Inline Functions", page 294.
macro	The **macro** special form is the primitive means of creating a macro. It gives a function spec a definition that is a macro definition rather than a actual function. A macro is not a function because it cannot be applied, but it *can* appear as the car of a form to be evaluated. Most macros are created with the more powerful **defmacro** special form.
defselect	The **defselect** special form defines a select-method function.
deff	Unlike the above special forms, **deff** does not create new functions. It simply serves as a hint to the editor that a function is being stored into a function spec here, and therefore if someone asks for the source code of the definition of that function spec, this is the place to look for it.
def	Unlike the above special forms, **def** does not create new functions. It simply serves as a hint to the editor that a function is being stored into a function spec here, and therefore if someone asks for the source code of the definition of that function spec, this is the place to look for it.

13.1.7 Lambda-List Keywords

This section documents all the keywords that can appear in the lambda-list (argument list) of a function, a macro, or a special form. See the section "Evaluating a Function Form", page 504. Some of them are allowed everywhere, while others are only allowed in one of these contexts; those are so indicated.

13.1.8 Declarations

Declarations are optional Lisp expressions that provide the Lisp system, typically the interpreter and the compiler, with information about your program, for example, documentation.

The special operator **declare** is the most common mechanism for making declarations. The other special operator, **zl:local-declare** should *not* be used for new code.

Many forms, such as **defun**, **defvar**, and **zl:defconst**, have declarative aspects. For example, **defun** tells the system that a function of a certain name and number of arguments is defined and where it is defined. **defvar** and **zl:defconst** tell the system that certain symbols are special.

13.1.9 How Programs Examine Functions

These functions take a function as argument and return information about that 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. (Note that a symbol is a function as well as a function spec).

13.1.10 How Programs Manipulate Definitions

A *definition* is a Lisp expression that appears in a source program file and has a name by which a user would like to refer to it. Definitions come in a variety of types. The main point of definition types is that two definitions with the same name and different types can exist simultaneously, but two definitions with the same name and the same type redefine each other when evaluated. Some examples of definition type symbols and special forms that define such definitions are:

Type symbol	*Type name in English*	*Special form names*
defun	function	**defun, defmacro, defmethod**
defvar	variable	**defvar, zl:defconst, defconstant**
defflavor	flavor	**defflavor**
zl:defstruct	structure	**zl:defstruct**

Things to note: More than one special form can define a given kind of definition. The name of the most representative special form is typically chosen as the type symbol. This symbol typically has a **si:definition-type-name** property of a string that acts as a prettier form of the name for people to read.

```
(defprop feature "Feature" si:definition-type-name)
(defprop defun "Function" si:definition-type-name)
```

record-source-file-name and related functions take a name and a type symbol as arguments. The editor understands certain definition-making special forms, and knows how to parse them to get out the name and the type. This mechanism has not yet been made user-extensible. Currently the editor assumes that any top-level form it does not know about that starts with "(def" must be defining a function (a definition of type **defun**) and assumes that the cadr of that form is the name of the function. The starting left parenthesis must be at the left margin (not indented) for the editor to recognize the "(def" form. Heuristics appropriate for **defun** are applied to this name if it is a list.

In general, a definition whose name is not a symbol and whose type is not **defun** does not work properly. This will be fixed in a future release.

The declaration **sys:function-parent** is of interest to users. The function with the same name is probably not of interest to users; it is part of the mechanism by which the Zmacs command Edit Definition (m-.) figures out what file to look in.

Example:

We have functions called "frobulators" that are stored on the property list of symbols and require some special bindings wrapped around their bodies. Frobulator definitions are not considered function definitions, because the name of the frobulator does not become defined as a Lisp function. Indeed, we could have a frobulator named **list** and Lisp's **list** function would continue to work. Instead we make a new definition type.

```
(defmacro define-frobulator (name arg-list &body body)
  `(progn
     (add-to-list-of-known-frobulators ',name)
     (record-source-file-name ',name 'define-frobulator)
     (defun (:property ,name frobulator) (self ,@arg-list)
       (declare (sys:function-parent ,name define-frobulator))
       (let (,(make-frobulator-bindings name arg-list))
         ,@body))))

(defprop define-frobulator "Frobulator" si:definition-type-name)
```

Here we would tell the editor how to parse **define-frobulator** if its parser were user-extensible. Because it is not, we rely on its heuristics to make `m-.` work adequately for frobulators.

Next we define a frobulator. This is not an interesting definition, for we do not actually know what the word "frobulate" means. We could always recast this example as a symbolic differentiator: We would define the + frobulator to return a list of + and the frobulations of the arguments, the * frobulator to return sums of products of factors and derivatives of factors, and so forth.

```
(define-frobulator list ()
  (frobulate-any-number-of-args self))
```

In **define-frobulator,** we call **record-source-file-name** so that when a file containing frobulator definitions is loaded, we know what file those definitions came from. Inside the function that is generated, we include a function-parent declaration because no definition of that function is apparent in any source file. The system takes care of doing **(record-source-file-name '(:property list frobulator) 'defun)**, as it always does when a function definition is loaded. Suppose an error occurs in a frobulator function – in the **list** example above, we might try to call **frobulate-any-number-of-args**, which is not defined – and we use the Debugger `c-E` command to edit the source. This is trying to edit **(:property list frobulator)**, the function in which we were executing. The definition that defines this function does not have that name; rather, it is named **list** and has type **define-frobulator.** The **sys:function-parent** declaration enables the editor to know that fact.

If your definition-making special form and your definition type symbol do not have the same name, you should define the special form's **zwei:definition-function-spec-type** property to be the definition type symbol. This helps the editor parse such special forms. This is useful when several special forms exist to make definitions of a single type.

For another example, more complicated but real, use **mexp** or the Zmacs command Macro Expand Expression (`c-sh-M`) to look at the macro expansion of:

```
(defstruct (foo :conc-name) one two)
```

The macro **sys:defsubst-with-parent** that it calls is just **defsubst** with a **sys:function-parent** declaration inside. It exists only because of a bug in an old implementation of **defsubst** that made doing it the straightforward way not work.

13.1.11 Encapsulations

The definition of a function spec actually has two parts: the *basic definition*, and *encapsulations*. The basic definition is what functions like **defun** create, and encapsulations are additions made by **trace**, **advise**, or **breakon** 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 done a second time, it replaces the old basic definition with a new one while leaving the encapsulations alone.

Only advanced users should ever need to use encapsulations directly via the primitives explained in this section. The most common things to do with encapsulations are provided as higher-level, easier-to-use features: **trace**, **advise**, and **breakon**.

The way the basic definition and the encapsulations are defined is that the actual definition of the function spec is the outermost encapsulation; this contains the next encapsulation, and so on. The innermost encapsulation contains the basic definition. The way this containing is done is as follows. An encapsulation is actually a function whose debugging info alist contains an element of the form:

```
(si:encapsulated-definition uninterned-symbol encapsulation-type)
```

You recognize a function to be an encapsulation by using **si:function-encapsulated-p**. An encapsulation is usually an interpreted function, but it can be a compiled function also, if the application that created it wants to compile it.

uninterned-symbol's function definition is the thing that the encapsulation contains, usually the basic definition of the function spec. Or it can be another encapsulation, which has in it another debugging info item containing another uninterned symbol. Eventually you get to a function that is not an encapsulation; it does not have the sort of debugging info item that encapsulations all have. That function is the basic definition of the function spec.

Literally speaking, the definition of the function spec is the outermost encapsulation, period. The basic definition is not the definition. If you are asking for the definition of the function spec because you want to apply it, the outermost encapsulation is exactly what you want. But the basic definition can be found mechanically from the definition, by following the debugging info alists. So it makes sense to think of it as a part of the definition. In regard to the function-defining special forms such as **defun**, it is convenient to think of the encapsulations as connecting between the function spec and its basic definition.

An encapsulation is created with the macro **si:encapsulate**.

You can test for an encapsulation with the function **si:function-encapsulated-p**.

It is possible for one function to have multiple encapsulations, created by different subsystems. In this case, the order of encapsulations is independent of the order in which they were made. It depends instead on their types. All possible encapsulation types have a total order and a new encapsulation is put in the right place among the existing encapsulations according to its type and their types.

Every symbol used as an encapsulation type must be on the list **si:encapsulation-standard-order**. In addition, it should have an **si:encapsulation-grind-function** property whose value is a function that **grindef** calls to process encapsulations of that type. This function need not take care of printing the encapsulated function, because **grindef** does that itself. But it should print any information about the encapsulation itself that the user ought to see. Refer to the code for the grind function for **advise** to see how to write one. See the special form **advise** in *Program Development Utilities*.

To find the right place in the ordering to insert a new encapsulation, it is necessary to parse existing ones. This is done with the function **si:unencapsulate-function-spec**.

13.1.11.1 Rename-Within Encapsulations

One special kind of encapsulation is the type **si:rename-within**. This encapsulation goes around a definition in which renamings of functions have been done.

How is this used?

If you define, advise, or trace **(:within foo bar)**, then **bar** gets renamed to **altered-bar-within-foo** wherever it is called from **foo**, and **foo** gets a **si:rename-within** encapsulation to record the fact. The purpose of the encapsulation is to enable various parts of the system to do what seems natural to the user. For example, **grindef** notices the encapsulation, and so knows to print **bar** instead of **altered-bar-within-foo**, when grinding the definition of **foo**.

Also, if you redefine **foo**, or trace or advise it, the new definition gets the same renaming done (**bar** replaced by **altered-bar-within-foo**). To make this work, everyone who alters part of a function definition should pass the new part of the definition through the function **si:rename-within-new-definition-maybe**.

13.2 Dynamic Closures

A *closure* is a type of Lisp functional object useful for implementing certain advanced access and control structures. Closures give you more explicit control over the environment, by allowing you to save the environment created by the entering of a dynamic contour (that is, a **lambda**, **do**, **prog**, **zl:progv**, **let**, or any of several other special forms), and then use that environment elsewhere, even after the contour has been exited.

13.2.1 What Is a Dynamic Closure?

We use a particular view of lambda-binding in this section because it makes it easier to explain what closures do. In this view, when a variable is bound, a new value cell is created for it. The old value cell is saved away somewhere and is inaccessible. Any references to the variable get the contents of the new value cell, and any **setq**'s change the contents of the new value cell. When the binding is undone, the new value cell goes away, and the old value cell, along with its contents, is restored.

For example, consider the following sequence of Lisp forms:

```
(setq a 3)

(let ((a 10))
  (print (+ a 6)))

(print a)
```

Initially there is a value cell for **a**, and the **setq** form makes the contents of that value cell be **3**. Then the **let** is evaluated. **a** is bound to **10**: the old value cell, which still contains a **3**, is saved away, and a new value cell is created with **10** as its contents. The reference to **a** inside the **let** evaluates to the current binding of **a**, which is the contents of its current value cell, namely **10**. So **16** is printed. Then the binding is undone, discarding the new value cell, and restoring the old value cell, which still contains a **3**. The final **print** prints out a **3**.

The form **(zl:closure** *var-list function***)**, where *var-list* is a list of special variables and *function* is any function, creates and returns a closure. When this closure is applied to some arguments, all the value cells of the variables on *var-list* are saved away, and the value cells that those variables had *at the time* **zl:closure** *was called* (that is, at the time the closure was created) are made to be the value cells of the symbols. Then *function* is applied to the arguments.

Here is another, lower level explanation. The closure object stores several things inside of it. First, it saves the *function*. Secondly, for each variable in *var-list*, it remembers what that variable's value cell was when the closure was created.

Then when the closure is called as a function, it first temporarily restores the value cells it has remembered inside the closure, and then applies *function* to the same arguments to which the closure itself was applied. When the function returns, the value cells are restored to be as they were before the closure was called.

Now, if we evaluate the form (assuming that **x** has been declared special):

```
(setq a
      (let ((x 3))
        (closure '(x) 'frob)))
```

what happens is that a new value cell is created for **x**, and its contents is an integer **3**. Then a closure is created, which remembers the function **frob**, the symbol **x**, and that value cell. Finally the old value cell of **x** is restored, and the closure is returned. Notice that the new value cell is still around, because it is still known about by the closure. When the closure is applied, say by doing **(funcall a 7)**, this value cell is restored and the value of **x** is **3** again. If **frob** uses **x** as a free variable, it sees **3** as the value.

A closure can be made around any function, using any form that evaluates to a function. The form could evaluate to a lambda expression, as in **'(lambda () cl:x)**, or to a compiled function, as would **(function (lambda () cl:x))**. In the example above, the form is **'frob** and it evaluates to the symbol **frob**. A symbol is also a good function. It is usually better to close around 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, then close around the definition of the symbol rather than the symbol itself. In the above example, that would be done by:

```
(closure '(x) (function frob))
```

Because of the way dynamic closures are implemented, the variables to be closed over must be declared special. This can be done with an explicit **declare**, with a special form such as **defvar**, or with **zl:let-closed**. In simple cases, a **declare** just inside the binding does the job. Usually the compiler can tell when a special declaration is missing, but in the case of making a closure the compiler detects this after already acting on the assumption that the variable is local, by which time it is too late to fix things. The compiler warns you if this happens.

In Symbolics Common Lisp's implementation of dynamic closures, lambda-binding of special variables never really allocates any storage to create new value cells. Value cells are created only by the **zl:closure** function itself, when they are needed. Thus, implementors of large systems need not worry about storage allocation overhead from this mechanism if they are not using dynamic closures.

Symbolics Common Lisp dynamic closures are not closures in the true sense, as they do not save the whole variable-binding environment; however, most of that environment is irrelevant, and the explicit declaration of which variables are to be closed allows the implementation to have high efficiency. They also allow you to explicitly choose for each variable whether it is to be bound at the point of call or bound at the point of definition (for example, creation of the closure), a choice which is not conveniently available in other languages. In addition, the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected.

Symbolics Common Lisp also offers lexical closures, which save the variable bindings of all accessible local and instance variables. Lexical closures do not affect the bindings of special variables. There is no function to create a lexical closure; one is created automatically wherever you use a function with captured free references. See the section "Kinds of Variables", page 495. See the section "Funargs and Lexical Closure Allocation", page 519.

The implementation of dynamic closures (which is not usually necessary for you to understand) involves two kinds of value cells. Every symbol has an *internal value cell*, which is where its value is normally stored. When a variable is closed over by a closure, the variable gets an *external value cell* to hold its value. The external value cells behave according to the lambda-binding model used earlier in this section. The value in the external value cell is found through the usual access mechanisms (such as evaluating the symbol, calling **zl:symeval**, and so on), because the internal value cell is made to contain an invisible pointer to the external value cell currently in effect. A symbol uses such an invisible pointer whenever its current value cell is a value cell that some closure is remembering; at other times, there is not an invisible pointer, and the value just resides in the internal value cell.

Most special variables that live in A-memory cannot be closed over.

13.2.2 Examples of the Use of Dynamic Closures

One thing we can do with dynamic closures is to implement a *generator*, which is a kind of function that is called successively to obtain successive elements of a sequence. We will implement a function **make-list-generator**, which takes a list and returns a generator that returns successive elements of the list. When it gets to the end it should return **nil**.

The problem is that in between calls to the generator, the generator must somehow remember where it is up to in the list. Since all of its bindings are undone when it is exited, it cannot save this information in a bound variable. It could save it in a global variable, but the problem is that if we want to have more than one list generator at a time, they all try to use the same global variable and get in each other's way.

Here is how we can use dynamic closures to solve the problem:

```
(defun make-list-generator (l)
  (declare (special l))
  (closure '(l)
           (function (lambda ()
                       (prog1 (car l)
                              (setq l (cdr l)))))))
```

Now we can make as many list generators as we like; they do not get in each other's way because each has its own (external) value cell for **l**. Each of these value cells was created when the **make-list-generator** function was entered, and the value cells are remembered by the closures. We could also use lexical closures to solve the same problem.

```
(defun make-list-generator (l)
  (function (lambda ()
              (prog1 (car l)
                     (setq l (cdr l))))))
```

The following example uses closures to create an advanced accessing environment:

```
(declare (special a b))

(defun foo ()
   (setq a 5))

(defun bar ()
   (cons a b))

(let ((a 1)
      (b 1))
   (setq x (closure '(a b) 'foo))
   (setq y (closure '(a b) 'bar)))
```

When the **let** is entered, new value cells are created for the symbols **a** and **b**, and two closures are created that both point to those value cells. If we do **(funcall x)**, the function **foo** is run, and it changes the contents of the remembered value cell of **a** to **5**. If we then do **(funcall F y)**, the function **bar** returns **(5 . 1)**. This shows that the value cell of **a** seen by the closure **y** is the same value cell seen by the closure **x**. The top-level value cell of **a** is unaffected.

To do this example with lexical closures, **foo** and **bar** would have to be defined with **flet** or **labels** so that they would share a lexical environment and contain captured free references to the same local variables **a** and **b**.

13.2.3 Dynamic Closure-Manipulating Functions

make-dynamic-closure *symbol-list function*	Creates a dynamic closure *function* over *symbol-list.*
closure-function *closure*	Returns the closed function *closure.*
symbol-value-in-closure *closure symbol*	Returns the binding of *symbol* in *closure.*
dynamic-closure-alist *closure*	Returns an alist of (*symbol* . *value*) pairs describing the bindings that the dynamic closure *closure* performs when it is called.
let-and-make-dynamic-closure *((variable value...)) function*	Binds *variable* to *value* and creates a closure over the *(variable value)* pairs, declaring them special in *function.*
copy-dynamic-closure *closure*	Creates and returns a list of all the variables in the dynamic closure *closure.*
dynamic-closure-variables *closure*	Creates and returns a list of all the variables in the dynamic closure *closure.*
boundp-in-closure *closure symbol*	Returns t if *symbol* is bound in the environment of *closure.*
makunbound-in-closure *closure symbol*	Makes *symbol* unbound in the environment of *closure.*
zl:closure *var-list function*	Creates and returns a dynamic closure of *function* over *varlist.*
zl:symeval-in-closure *closure symbol*	Like **symbol-value-in.**
zl:set-in-closure *closure symbol x*	Sets the binding of *symbol* to *x* in the environment *closure.*
zl:locate-in-closure *closure symbol*	Returns the location of the place in *closure* where the saved value of *symbol* is stored.
zl:closure-alist *closure*	Like **dynamic-closure-alist.**

zl:let-closed *((variable value...)) function*	
	Like **let-and-make-dynamic-closure.**
zl:copy-closure *closure*	Like **copy-dynamic-closure.**
zl:closure-variables *closure*	Like **dynamic-closure-variables.**

A note about all of the *xxx*-**in-closure** functions (**set-**, **symeval-**, **boundp-**, and **makunbound-**): if the variable is not directly closed over, the variable's value cell from the global environment is used. That is, if closure A closes over closure B, *xxx*-**in-closure** of A does not notice any variables closed over by B.

14. Predicates

A *predicate* is a function that tests for some condition involving its arguments and returns some non-**nil** value if the condition is true, or the symbol **nil** if it is not true. Predicates such as **and**, **zl:member** and **special-form-p** return non-**nil** values when the condition is true, while predicates such as **numberp**, **zl:listp** and **functionp** return the symbol **t** if the condition is true. An example of the non-**nil** return value is the predicate **special-form-p**. It returns a function that can be used to evaluate the special form.

By convention, the names of predicates usually end in the letter "p" (which stands for "predicate"). The way the "p" is added to the end of the predicate is dependent on whether or not there is an existing hyphen in the name. For instance, the list predicate is **zl:listp**, while the predicate that checks for compiled functions is **compiled-function-p**.

The summary tables below group predicates by function for a quick overview. See the alphabetized Dictionary of Lisp functions for a full description of individual predicates.

The following predicates test data types. These predicates return **t** if the argument is of the type indicated by the name of the function, **nil** if it is of some other type.

14.1 Numeric Type-checking Predicates

These predicates test a number to see if it belongs to a given type. General type-checking functions such as **typep** and **subtypep** can also be used to determine relationships within the hierarchy of numeric types and for similar purposes. For more on these functions: See the section "Determining the Type of an Object", page 79.

complexp *object*	Tests for complex number.
floatp *object*	Tests for floating-point number of any precision.
integerp *object*	Tests for integer.
numberp *object*	Tests for number of any type.
rationalp *object*	Tests for rational number.
sys:double-float-p *object*	Tests for double-precision floating-point number.

sys:single-float-p *object*	Tests for single-precision floating-point number.
zl:bigp *object*	Tests for bignum.
zl:fixnump *object*	Tests for fixnum.
zl:fixp *object*	Tests for integer (same as **integerp**).
zl:flonump *object*	Tests for single-precision floating-point number (same as **sys:single-float-p**).
zl:rationalp *object*	Tests for ratio.

14.2 Array Type-checking Predicates

adjustable-array-p *array*	tests if the array size is dynamically changeable
array-has-fill-pointer-p *array*	tests if *array* has a fill pointer (array must be one-dimensional)
array-has-leader-p *array*	tests if *array* has a leader
array-in-bounds-p *array* &rest *subscripts*	tests whether all of the subscripts are legal for *array*
arrayp *array*	tests if *array* is any type of array
sys:array-displaced-p *array*	tests if *array* is any kind of displaced array (including indirect)
sys:array-indexed-p *array*	tests if *array* is an indirect array with an index-offset
sys:array-indirect-p *array*	tests if *array* is an indirect array

14.3 Vector Type-checking Predicates

bit-vector-p *object*	tests if *object* is a one-dimensional array of bits
simple-bit-vector-p *object*	tests if *object* is a simple bit-vector
simple-vector-p *object*	tests if *object* is a simple vector
vectorp *object*	tests if *object* is a one-dimensional array

14.4 Character Type-checking Predicates

alpha-char-p *char*	tests if *char* is an alphabetic character
alphanumericp *char*	tests if *char* is either alphabetic or numeric
char-fat-p *char*	tests if *char* is a character that has non-zero *bits* or *font* attribute
characterp *object*	tests if *object* is a character
digit-char-p *char* &optional (*radix* 10)	tests if *char* is a digit of the radix specified by *radix* and returns a non-negative integer that is the "weight" of *char* in *radix*
formatting-char-p *char*	tests if *char* is invisible and should be printed as it's name rather than itself (eg. #\Line)
graphic-char-p *char*	tests if *char* is a printing character
standard-char-p *char*	tests if *char* is a character with zero *bits* and *font* attribute
mouse-char-p *char*	tests if *char* is a mouse-character representing the clicking of a mouse button

14.5 Character Case-checking Predicates

both-case-p *char*	tests if *char* is a character for which there is both an uppercase and corresponding lowercase character equivalent
lower-case-p *char*	tests if *char* is a lowercase character
upper-case-p *char*	tests if *char* is an uppercase character

14.6 Input/Output Type-checking Predicates

input-stream-p *stream*	tests if *stream* can handle input operations
output-stream-p *stream*	tests if *stream* can handle output operations
pathnamep *object*	tests if *object* is a pathname
readtablep *object*	tests if *object* is a readtable
streamp *object*	tests if *object* is a stream

14.7 String Type-Checking Predicates

These predicates test whether an object is a string of the recognized string types. The general type-checking predicate **typep** can also be used to test for strings. See the section "Determining the Type of an Object", page 79.

simple-string-p *object*	Returns **t** if *object* is a simple string array (one with no fill pointer and no displacement), and **nil** otherwise; accepts any object as an argument.
string-char-p *char*	Returns **t** if *char* can be stored into a thin string (that is, if it is a standard character; returns **nil** otherwise. Accepts character argument only.

string-fat-p *string*	Returns **t** if *string* is an array of fat characters, and **nil** otherwise. Accepts a string argument only.
stringp *object*	Returns **t** if *object* is either type of string, and **nil** otherwise; accepts any object as an argument.

14.8 Non-numeric Data Type-checking Predicates

atom *object*	tests if *object* is not a cons
consp *object*	tests if *object* is a cons
instancep *object*	tests if *object* is an instance of a flavor
zl:listp *object*	tests if *object* is a cons or the empty list
locativep *object*	tests if *object* is a locative
zl:nlistp *object*	tests if *object* is anything but a cons (same as **atom**)
nsymbolp *object*	tests if *object* is not a symbol
symbolp *object*	tests if *object* is a symbol

14.9 Other Type-checking Predicates

commonp *object*	tests if *object* is any valid Common Lisp data type
compiled-function-p *object*	tests if *object* is any compiled code object
constantp *object*	tests if *object* always evaluates to the same thing
errorp *object*	tests if *object* is an error object

functionp *object*	tests if *object* is a function
named-structure-p *object*	tests if *object* is a named structure and returns *object*'s named structure symbol
special-form-p *symbol*	tests if *symbol* is a globally-named special form
subtypep *type1 type2*	tests if *typ1* is definitely a subtype of type2 (for exact return values see the dictionary entry)
zl:typep *object type*	tests if *object* is of type *type*
zl:closurep *object*	tests if *object* is a closure
zl:subrp *arg*	tests if *arg* is a compiled code object
zl:typep *arg* &optional *type*	tests for type specified (two arguments) and returns argument type (one argument)

The following predicates test properties of numbers. These predicates return **t** if the argument has the property indicated by the name of the function, **nil** if it does not.

14.10 Numeric Property-checking Predicates

evenp *integer*	tests for even integers
oddp *integer*	tests for odd integers
minusp *number*	tests if number is less than zero
plusp *number*	tests if number is greater than zero
zerop *number*	tests if a number is zero
zl:signp *test number*	tests if the sign of *number* is *test*

The following predicates perform comparisons on numbers. These predicates return **t** if the arguments pass the test indicated by the name of the function, **nil** if they do not.

14.11 Numeric Comparison Functions

Function	*Synonyms*	*Comparison/Returned Value*
≠ *number* &rest *numbers*	/=	Not equal
< *number* &rest *more-numbers*	**zl:lessp**	Less-than
≤ *number* &rest *more-numbers*	<=	Less-than-or-equal
= *number* &rest *more-numbers*		Equal
> *number* &rest *more-numbers*	**zl:greaterp**	Greater-than
≥ *number* &rest *more-numbers*	>=	Greater-than-or-equal
max *number* &rest *more-numbers*		Greatest of its arguments
min *number* &rest *more-numbers*		Least of its arguments

These predicates test characters using implementation-dependent total ordering on code attributes. These tests are case sensitive.

14.12 Case-Sensitive Character Comparison Predicates

char/= *char* &rest *more-chars*	Not the same
char≠ *char* &rest *more-chars*	Not the same (same as **char/=**)
char< *char* &rest *more-chars*	Less-than
char<= *char* &rest *more-chars*	Less-than-or-equal

char≤ *char* &rest *more-chars*	Less-than-or-equal (same as **char<=**)
char= *char* &rest *more-chars*	The same
char> *char* &rest *more-chars*	Greater-than
char>= *char* &rest *more-chars*	Greater-than-or-equal
char≥ *char* &rest *more-chars*	Greater-than-or-equal (same as **char>=**)

These predicates test characters using a different ordering scheme that accounts for differences in font information, but ignores differences in bits attributes and case.

14.13 Case-Insensitive Character Comparison Predicates

char-equal *char* &rest *more-chars*	like **char=**
char-not-equal *char* &rest *more-chars*	like **char/=**
char-lessp *char* &rest *more-chars*	like **char<**
char-greaterp *char* &rest *more-chars*	like **char>**
char-not-greaterp *char* &rest *more-chars*	like **char<=**
char-not-lessp *char* &rest *more-chars*	like **char>=**

14.14 Case-Sensitive String Comparison Predicates

These predicates compare two strings or substrings of them, exactly, depending on all fields including character style, and alphabetic case. See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

The keywords (*:start1* **0**) and (*:start2* **0**) specify the character position (counting from 0) from which to *begin* the comparison; the keywords (*:end1* **nil**) and (*:end2*

nil) specify the character position immediately *after* the end of the comparison. The start arguments default to **0** (compare strings in their entirety); the end arguments default to the length of the string. The start arguments must be ≤ the end arguments.

The predicates compare the strings in dictionary order. They return either the symbol **nil** or, generally, the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

For the Zetalisp versions of these predicates, the optional arguments, *idx1* and *idx2* specify the start point for the comparison, while *lim1* and *lim2* specify the character immediately after the end of the comparison. These Zetalisp predicates generally return either **t** or **nil**.

string= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if corresponding characters in the strings are identical or if the strings are unequal in length.

string≠ *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the characters in the two strings are not identical (same as **string/=**).

string/= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
A synonym of **string≠**.

string< *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char<**, or if *string1* is a proper substring of *string2*.

string≤ *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char≤**, or if *string1* is a substring of *string2* (same as **string<=**).

string<= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
A synonym of **string≤**.

string> *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char>**, or if *string2* is a proper substring of *string1*.

string≥ *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Tests if the first characters that differ between *string1* and *string2* are **char≥**, or if *string2* is a substring of *string1* (same as **string>=**).

string>= *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
A synonym of **string≥**.

string-exact-compare *string1 string2* &key (I[start1] **0**) (*start2* **0**) (*end1* **nil**) (*end2* **nil**)
Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*.

sys:%string= *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison.

sys:%string-exact-compare *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison. Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*.

zl:string= *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string=**, but returns **t** or **nil**.

zl:string≠ *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string≠**, but returns **t** or **nil**.

zl:string< *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string<**, but returns **t** or **nil**.

zl:string> *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string>**, but returns **t** or **nil**.

zl:string≤ *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string≤**, but returns **t** or **nil**.

zl:string≥ *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string≥**, but returns **t** or **nil**.

zl:string-exact-compare *string1 string2* &optional *idx1 idx2 lim1 lim2*
Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. Use **string-exact-compare** instead.

14.15 Case-Insensitive String Comparison Predicates

These predicates test strings ignoring character case and character style. See the section "Case-Sensitive and Case-Insensitive String Comparisons", page 226.

The keywords (*:start1* **0**) and (*:start2* **0**) specify the character position (counting from 0) from which to *begin* the comparison; the keywords (*:end1* **nil**) and (*:end2* **nil**) specify the character position immediately *after* the end of the comparison. The start arguments default to **0** (compare strings in their entirety); the end arguments default to the length of the string. The start arguments must be ≤ the end arguments.

The predicates compare the strings in dictionary order. They return either the symbol **nil** or, generally, the position within the strings of the first character at which the strings fail to match; this index is equivalent to the length of the longest common portion of the strings.

For the Zetalisp versions of these predicates, the optional arguments *idx1* and *idx2* specify the start point for the comparison, while *lim1* and *lim2* specify the character immediately after the end of the comparison. These Zetalisp predicates generally return either **t** or **nil.**

string-equal *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string=**.

string-not-equal *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string/=**.

string-lessp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string<**.

string-greaterp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string>**.

string-not-greaterp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string<=**.

string-not-lessp *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string>=**.

string-compare *string1 string2* &key (*:start1* **0**) (*:end1* **nil**) (*:start2* **0**) (*:end2* **nil**)
Case-insensitive version of **string-exact-compare.**

sys:%string-equal *string1 index1 string2 index2 count*
A low-level, possibly more efficient string comparison. Case-insensitive version of **sys:%string=**.

sys:%string-compare *string1 index1 string2 index2 count*

A low-level, possibly more efficient string comparison. Case-insensitive version of **sys:%string-exact-compare**.

zl:string-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string=**. Use **string-equal** instead.

zl:string-not-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string≠**. Like **string-not-equal** but returns **t** or **nil**.

zl:string-lessp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string<**. Like **string-lessp** but returns **t** or **nil**.

zl:string-greaterp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string>**. Like **string-greaterp** but returns **t** or **nil**.

zl:string-not-lessp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string-not-lessp** but returns **t** or **nil**.

zl:string-not-greaterp *string1 string2* &optional *idx1 idx2 lim1 lim2*
Like **string-not-greaterp** but returns **t** or **nil**.

zl:string-compare *string1 string2* &optional *idx1 idx2 lim1 lim2*
Case-insensitive version of **zl:string-exact-compare**. Use **string-compare** instead.

14.16 Comparison-performing Predicates

eq *x y*	tests for identical object (implementationally)
eql *x y*	tests for identical object (conceptually)
zl:equal *x y*	tests for similar objects
equalp *x y*	tests for similar objects (differs from **zl:equal** on arrays and characters)
neq *x y*	not **eq**
not *x*	tests for the symbol **nil**
null *object*	tests for the empty list **nil** ()

alphalessp *string1 string2*	like **zl:string-lessp**, but also works on numbers, lists and other objects
zl:samepnamep *sym1 sym2*	like **zl:string=**; tests if the two symbols have **zl:string=** printed representations

The following predicates perform bit-testing checks on bit-vectors. These predicates return **t** or **nil**.

14.17 Predicates for Testing Bits in Integers

logbitp *index integer*	Returns true if *index* bit in *integer* (the bit whose weight is 2^{index}) is a one-bit
logtest *integer1 integer2*	Returns true if any 1-bits in *integer1* are 1-bits in *integer2*
zl:bit-test *x y*	Returns true if any 1 bits in *x* are 1 bits in *y* (same as **logtest**)

These predicates perform tests on flavors.

14.18 Flavor Predicates

flavor-allows-init-keyword-p *flavor-name keyword*	tests if keyword is a valid init option for *flavor-name* and returns the component flavor name that handles the keyword or the symbol **nil** if it is not a valid init option
operation-handled-p *object message-name* &rest *arguments*	tests if the flavor associated with *object* has a method defined for *message-name*
zl:instance-variable-boundp *var*	tests if *var* is a bound instance variable

These predicates perform tests on parts of packages.

14.19 Package Predicates

boundp *sym*	tests if *sym* is bound
fboundp *sym*	tests if *sym*'s function cell is not empty
location-bound-p *location*	tests if cell pointed to by locative pointer *location* is bound
variable-boundp *variable*	tests if local, special, or instance variables are bound
fdefinedp *function-spec*	tests if *function-spec* has a definition

These predicates perform tests on lists and tables.

14.20 List and Table Predicates

endp *object*	tests for the end of a list
subsetp *list1 list2* &key **:test :test-not :key**	tests if every element of *list1* is in *list2*
tree-equal *x y* &key **:test :test-not**	tests for isomorphic trees with identical leaves

14.21 Miscellaneous Predicates

time-elapsed-p *increment initial-time* &optional *(final-time* **(zl:time)***)*	tests if *increment* 60ths of a second have elapsed between *initial-time* and *final-time*
time-lessp *time1 time2*	tests if *time1* is earlier than *time2*

15. Macros

15.1 Introduction to Macros

If **eval** is handed a list whose car is a symbol, then **eval** inspects the definition of the symbol to find out what to do. If the definition is a macro, the definition contains an expander function. **eval** applies the expander function to two arguments, the form that **eval** is trying to evaluate and an object representing the lexical environment. The expander function returns a new form and **eval** evaluates that in lieu of the original form.

Here is a simple example. Suppose the definition of the symbol **first** is:

```
(defmacro first (x &environment env)
          (list 'car x)))
```

first is a macro. What happens if we try to evaluate the form **(first '(a b c))**? **eval** sees that it has a list whose car is a symbol (namely, **first**), so it looks at the definition of the symbol and sees that the definition is a macro.

eval gets the macro's *expander function*, and calls it providing as arguments the original form that **eval** was handed and the lexical environment. So it calls **(lambda (x env) (list 'car (cadr x)))** with arguments **(first '(a b c))** and the lexical environment. Whatever this returns is the *expansion* of the macro call. It is evaluated in place of the original form.

In this case, **x** is bound to **(first '(a b c))**, **(cadr x)** evaluates to **'(a b c)**, and **(list 'car (cadr x))** evaluates to **(car '(a b c))**, which is the expansion. **eval** now evaluates the expansion. **(car '(a b c))** returns **a**, and so the result is that **(first '(a b c))** returns **a**.

What have we done? We have defined a macro called **first**. What the macro does is to *translate* the form to some other form. Our translation is very simple – it just translates forms that look like **(first** *x***)** into **(car** *x***)**, for any form *x*. We can do much more interesting things with macros, but first we show how to define a macro.

The primitive special form for defining macros is **macro**. A macro definition looks like this:

```
(macro name (form env)
   body)
```

name can be any function spec. *form* and *env* must be variables. *body* is a sequence of Lisp forms that expand the macro; the last form should return the expansion. **defmacro** is usually preferred in practice.

To define our **first** macro, we would say:

```
(macro first (x ignore)
    (list 'car (cadr x)))
```

Here are some more simple examples of macros. Suppose we want any form that looks like **(addone** *x***)** to be translated into **(plus 1** *x***)**. To define a macro to do this we would say

```
(macro addone (x ignore)
   (list 'plus '1 (cadr x))))
```

Now say we wanted a macro that would translate **(increment** *x***)** into **(setq** *x* **(1+** *x***)**. This would be:

```
(macro increment (x ignore)
    (list 'setq (cadr x) (list '1+ (cadr x)))
```

Of course, this macro is of limited usefulness, because the form in the *cadr* of the **increment** form should be a symbol. If you tried **(increment (car x))**, it would be translated into **(setq (car x) (1+ (car x)))**, and **setq** would complain. (If you are interested in how to fix this problem: See the macro **setf** in *Symbolics Common Lisp: Language Dictionary*. However, this is irrelevant to how macros work.)

As you can see, macros are very different from functions. A function would not be able to tell what kind of subforms are in a call to itself; they get evaluated before the function ever sees them. However, a macro can look at the whole form and see what it is doing. Macros are not functions; if **first** is defined as a macro, it is not meaningful to apply **first** to arguments. A macro does not take arguments at all; its expander function takes a Lisp form and turns it into another Lisp form.

The purpose of functions is to *compute*; the purpose of macros is to *translate*. Macros are used for a variety of purposes, the most common being extensions to the Lisp language. For example, Lisp is powerful enough to express many different control structures, but it does not provide every control structure anyone might ever possibly want. Instead, if you want some kind of control structure with a syntax that is not provided, you can translate it into some form that Lisp *does* know about.

For example, you might want a limited iteration construct that increments a variable by one until it exceeds a limit (like the FOR statement of the BASIC language). You might want it to look like:

```
(for a 1 100 (print a) (print (* a a)))
```

To get this, you could write a macro to translate it into:

```
(do a 1 (1+ a) (> a 100) (print a) (print (* a a)))
```

A macro to do this could be defined with:

```
(macro for (x ignore)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

Now you have defined your own new control structure primitive, and it will act just as if it were a special form provided by Lisp itself.

A macro definition is represented as a list whose first element is the symbol **special**, whose second element is the expander function, and whose third element is missing or **nil**. (The third element is used by a special form definition.) **eval** applies the function to the form it was originally given, takes the expansion that is returned, and evaluates that in lieu of the original form.

Example:

```
(special (lambda (x any)
           (list 'car (cadr x))))
```

One useful function is **macro-function**. This function tests whether its argument is the name of a macro.

15.2 Aids for Defining Macros

The main problem with the following definition is that it is verbose and clumsy:

```
(macro for (x ignore)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

If it is that hard to write a macro to do a simple specialized iteration construct, you might wonder how anyone could write macros of any real sophistication.

There are two things that make the definition so inelegant. One is that you must write things like **(cadr x)** and **(cddddr x)** to refer to the parts of the form you want to do things with. The other problem is that the long chains of calls to the **list** and **cons** functions are very hard to read.

Two features are provided to solve these two problems. The **defmacro** macro solves the former, and the "backquote" (‘) reader macro solves the latter.

15.2.1 defmacro

Instead of referring to the parts of our form by **(cadr x)** and such, we would like to give names to the various pieces of the form, and somehow have the **(cadr x)** automatically generated. This is done by a macro called **defmacro**. It is easiest to explain what **defmacro** does by showing an example. Here is how you would write the **for** macro using **defmacro**:

```
(defmacro for (var lower upper . body)
  (cons 'do
        (cons var
              (cons lower
                    (cons (list '1+ var)
                          (cons (list '> var upper)
                                body))))))
```

(var lower upper . body) is a *pattern* to match against the body of the form (to be more precise, to match against the cdr of the first argument to the macro's expander function). If **defmacro** tries to match the following two lists:

```
(var lower upper . body)
(a 1 100 (print a) (print (* a a)))
```

var gets bound to the symbol **a**, **lower** to the integer **1**, **upper** to the integer **100**, and **body** to the list **((print a) (print (* a a)))**. Then inside the body of the **defmacro**, **var**, **lower**, **upper**, and **body** are variables, bound to the matching parts of the macro form.

defmacro is a general-purpose macro-defining macro. A **defmacro** form looks like:

(defmacro *name pattern* . *body*)

The *pattern* can be anything made up out of symbols and conses. It is matched against the body of the macro form; both *pattern* and the form are **car**'ed and **cdr**'ed identically, and whenever a non-**nil** symbol occurs in *pattern*, the symbol is bound to the corresponding part of the form. If the corresponding part of the form is **nil**, it goes off the end of the form. **&optional**, **&rest**, **&key**, and **&body** can be used to indicate where optional pattern elements are allowed.

All of the symbols in *pattern* can be used as variables within *body*. *name* is the name of the macro to be defined; it can be any function spec. See the section "Function Specs", page 251. *body* is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

defmacro could have been defined in terms of **destructuring-bind** as follows, except that the following is a simplified example of **defmacro** showing no error-checking and omitting the **&environment** and **&whole** features.

&whole is followed by *variable*, which is bound to the entire macro-call form or subform. *variable* is the value that the macro-expander function receives as its first argument. **&whole** is allowed only in the top-level pattern, not in inside patters.

&environment is followed by *variable*, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment.

```
(defmacro defmacro (name pattern &body body)
  `(macro ,name (form env)
     (destructuring-bind ,pattern (cdr form)
       . ,body)))
```

See the section "&-Keywords Accepted By **defmacro**", page 312.

See the special form **destructuring-bind** in *Symbolics Common Lisp: Language Dictionary.*

Note that the pattern need not be a list the way a lambda-list must. In the above example, the pattern was a "dotted list", since the symbol **body** was supposed to match the cddddr of the macro form. If we wanted a new iteration form, like **for** except that our example would look like:

```
(for a (1 100) (print a) (print (* a a)))
```

(just because we thought that was a nicer syntax), then we could do it merely by modifying the pattern of the **defmacro** above; the new pattern would be **(var (lower upper) . body)**.

Here is how we would write our other examples using **defmacro**:

```
(defmacro first (the-list)
    (list 'car the-list))

(defmacro addone (form)
   (list 'plus '1 form))

(defmacro increment (symbol)
   (list 'setq symbol (list '1+ symbol)))
```

All of these are very simple macros and have very simple patterns, but they show that we can replace **(cadr x)** with a readable mnemonic name such as **the-list** or **symbol**, which makes the program clearer, and enables documentation facilities such as the **arglist** function to describe the syntax of the special form defined by the macro.

There is another version of **defmacro** that defines displacing macros. See the section "Displacing Macros", page 311. **defmacro** has other, more complex features. See the section "&-Keywords Accepted By **defmacro**", page 312. A way to define local macros is discussed elsewhere. See the special form **macrolet**, page 525.

15.2.2 Backquote

Now we deal with the other problem: the long strings of calls to **cons** and **list**. This problem is relieved by introducing some new characters that are special to the Lisp reader. Just as the single-quote character makes it easier to type things of the form **(quote** *x***)**, some more new special characters make it easier to type forms that create new list structure. The functionality provided by these characters is called the *backquote* facility.

The backquote facility is used by giving a backquote character (‘), followed by a form. If the form does not contain any use of the comma character, the backquote acts just like a single quote: it creates a form that, when evaluated, produces the form following the backquote. For example:

```
'(a b c) => (a b c)
`(a b c) => (a b c)
```

So in the simple cases, backquote is just like the regular single-quote macro. The way to get it to do interesting things is to include a comma somewhere inside the form following the backquote. The comma is followed by a form, and that form gets evaluated even though it is inside the backquote. For example:

```
(setq b 1)
`(a b c)  => (a b c)
`(a ,b c) => (a 1 c)
`(abc ,(+ b 4) ,(- b 1) (def ,b)) => (abc 5 0 (def 1))
```

In other words, backquote quotes everything *except* things preceded by a comma; those things get evaluated.

A list following a backquote can be thought of as a template for some new list structure. The parts of the list that are preceded by commas are forms that fill in slots in the template; everything else is just constant structure that appears in the result. This is usually what you want in the body of a macro; some of the form generated by the macro is constant, the same thing on every invocation of the macro. Other parts are different every time the macro is called, often being functions of the form that the macro appeared in (the "arguments" of the macro). The latter parts are the ones for which you would use the comma. Several examples of this use follow.

When the reader sees the **‘(a ,b c)** it is actually generating a form such as **(list 'a b 'c)**. The actual form generated might use **list, cons, append**, or whatever might be a good idea; you should never have to concern yourself with what it actually turns into. All you need to care about is what it evaluates to. Actually, it does not use regular functions such as **cons** and **list**, but uses special ones instead so that the grinder can recognize a form that was created with the backquote syntax, and print it using backquote so that it looks like what you typed in. You should never write any program that depends on this, anyway, because backquote makes no guarantees about how it does what it does. In particular, in some circumstances it might decide to create constant forms that cause sharing of list structure at run time, or it might decide to create forms that create new list structure at run time. For example, if the reader sees **'(r . ,nil)**, it might produce the same thing as **(cons 'r nil)**, or **'(r . nil)**. Be careful that your program does not depend on one of these.

The following examples might make this behavior clearer. Here is how we would write our three simple macros using both the **defmacro** and backquote facilities.

```
(defmacro first (the-list)
    '(car ,the-list))

(defmacro addone (form)
   '(plus 1 ,form))

(defmacro increment (symbol)
   '(setq ,symbol (1+ ,symbol)))
```

Finally, to demonstrate how easy it is to define macros with these two facilities, here is the final form of the **for** macro.

```
(defmacro for (var lower upper . body)
  '(do ,var ,lower (1+ ,var) (> ,var ,upper) . ,body))
```

Look at how much simpler that is than the original definition. Also, look how closely it resembles the code it is producing. The functionality of the **for** stands out when written this way.

If a comma inside a backquote form is followed by an at-sign character (@), it has a special meaning. The **,@** should be followed by a form whose value is a *list*; then each of the elements of the list is put into the list being created by the backquote. In other words, instead of generating a call to the **cons** function, backquote generates a call to **append**. For example, if **a** is bound to **(x y z)**, then **'(1 ,a 2)** would evaluate to **(1 (x y z) 2)**, but **'(1 ,@a 2)** would evaluate to **(1 x y z 2)**.

Here is an example of a macro definition that uses the "**,@**" construction. Suppose you wanted to extend Lisp by adding a kind of special form called **repeat-forever**, which evaluates all of its subforms repeatedly. One way to implement this would be to expand:

```
(repeat-forever form1 form2 form3)
```

into:

```
(prog ()
    a form1
      form2
      form3
      (go a))
```

You could define the macro by:

```
(defmacro repeat-forever (&body body)
       `(prog ()
            a ,@body
              (go a)))
```

A similar construct is ",." (comma, dot). This means the same thing as ",@" except that the list that is the value of the following form can be freely smashed; backquote uses **nconc** rather than **append**. This should of course be used with caution.

Backquote does not make any guarantees about what parts of the structure it shares and what parts it copies. You should not do destructive operations such as **nconc** on the results of backquote forms such as:

```
`(,a b c d)
```

since backquote might choose to implement this as:

```
(cons a '(b c d))
```

and **nconc** would smash the constant. On the other hand, it would be safe to **nconc** the result of:

```
`(a b ,c ,d)
```

since there is nothing this could expand into that does not involve making a new list, such as:

```
(list 'a 'b c d)
```

Backquote of course guarantees not to do any destructive operations (**rplaca**, **rplacd**, **nconc**) on the components of the structure it builds, unless the **,.** syntax is used.

Advanced macro writers sometimes write "macro-defining macros": forms that expand into forms that, when evaluated, define macros. In such macros it is often useful to use nested backquote constructs. The following example illustrates the use of nested backquotes in the writing of macro-defining macros.

This example is a very simple version of **defstruct**. You should first understand the basic description of **defstruct** before proceeding with this example. The

defstruct below does not accept any options, and allows only the simplest kind of items; that is, it only allows forms like:

```
(defstruct (name)
     item1
     item2
     item3
     item4
     ...)
```

We would like this form to expand into:

```
(progn
 (defmacro item1 (x)
      `(aref ,x 0))
 (defmacro item2 (x)
      `(aref ,x 1))
 (defmacro item3 (x)
      `(aref ,x 2))
 (defmacro item4 (x)
      `(aref ,x 3))
 ...)
```

The meaning of the **(progn ...)** is discussed in another section. See the section "Macros Expanding Into Many Forms", page 301. Here is the macro to perform the expansion:

```
(defmacro defstruct ((name) . items)
     (do ((item-list items (cdr item-list))
          (ans nil)
          (i 0 (1+ i)))
         ((null item-list)
          `(progn . ,(nreverse ans)))
       (setq ans
             (cons `(defmacro ,(car item-list) (x)
                              `(aref ,x ,',i))
                   ans))))
```

The interesting part of this definition is the body of the (inner) **defmacro** form:

```
`(aref ,x ,',i)
```

Instead of using this backquote construction, we could have written:

```
(list 'aref x ,i)
```

That is, ",'," acts like a comma that matches the outer backquote, while "," preceding the "**x**" matches with the inner backquote. Thus, the symbol **i** is evaluated when the **defstruct** form is expanded, whereas the symbol **x** is evaluated when the accessor macros are expanded.

Backquote can be useful in situations other than the writing of macros. Whenever there is a piece of list structure to be consed up, most of which is constant, the use of backquote can make the program considerably clearer.

15.3 Inline Functions

An inline function is a function that is open-coded by the compiler. It is like any other function when applied, but it can be expanded instead, and in that regard resembles a macro. The primitive way to do this is with (**proclaim** '(**inline foo**)).

defsubst is an easier way to define inline functions. It is used just like **defun** and does almost the same thing.

(defsubst *name lambda-list* . *body*)

defsubst defines a function that executes identically to the one that a similar call to **defun** would define. The difference comes when a function that *calls* this one is compiled. Then, the call is open-coded by substituting the inline function's definition into the code being compiled. Such a function is called an **inline** function. For example, if we define:

```
(defsubst square (x) (* x x))

(defun foo (a b) (square (+ a b)))
```

then if **foo** is used interpreted, **square** works just as if it had been defined by **defun**. If **foo** is compiled, however, the squaring is substituted into it and it compiles just like:

```
(defun foo (a b) (* (+ a b) (+ a b)))
```

square could have been defined as:

```
(proclaim ((inline square (x) (* x x))

(defun foo ...)
```

See the declaration **inline** in *Symbolics Common Lisp: Language Dictionary.*

A similar **square** could be defined as a macro, with:

```
(defmacro square (x) `(* ,x ,x))
```

When the compiler open-codes an **inline** function, it binds the argument variables to the argument values with **let**, so they get evaluated only once and in the right order. Then, when possible, the compiler optimizes out the variables. In general, anything that is implemented as an **inline** function can be reimplemented as a macro, just by changing the **defsubst** to a **defmacro** and putting in the appropriate backquote and commas, except that this does not get the simultaneous guarantee of argument evaluation order and generation of optimal code with no unnecessary temporary variables. The disadvantage of macros is that they are not functions, and so cannot be applied to arguments. Their advantage is that they can do much more powerful things than **inline** functions can. This is also a disadvantage since macros provide more ways to get into trouble. If something can be implemented either as a macro or as an **inline** function, it is generally better to make it an **inline** function.

As with **defun**, *name* can be any function spec, but you get the "**subst**" effect only when *name* is a symbol.

The difference between an **inline** function and one not declared inline is the way the calls to them 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 apply the function to those values. A call to an **inline** function is compiled as an *open subroutine*; the compiler incorporates the body forms of the **inline** function into the function being compiled, substituting the argument forms for references to the variables in the function's *lambda-list*.

15.4 Symbol Macros

A symbol macro translates a symbol into a substitute form. When the Lisp evaluator is given a symbol, it checks whether the symbol has been defined as a symbol macro. If so, it evaluates the symbol's replacement form instead of the symbol itself.

define-symbol-macro *name form* defines a symbol macro. *name* is a symbol to be defined as a symbol macro. *form* is a Lisp form to be substituted for the symbol when the symbol is evaluated. A symbol macro is more like an inline function than a macro: *form* is the form to be substituted for the symbol, not a form whose evaluation results in the substitute form.

Example:

```
(define-symbol-macro foo (+ 3 bar))
(setq bar 2)
foo => 5
```

A symbol defined as a symbol macro cannot be used in the context of a variable. You cannot use **setq** on it, and you cannot bind it. You can use **setf** on it: **setf** substitutes the replacement form, which should access something, and expands into the appropriate update function.

For example, suppose you want to define some new instance variables and methods for a flavor. You want to test the methods using existing instances of the flavor.

For testing purposes, you might use hash tables to simulate the instance variables, using one hash table per instance variable with the instance as the key. You could then implement an instance variable **x** as a symbol macro:

```
(defvar x-hash-table (make-hash-table))
(define-symbol-macro x (send x-hash-table :get-hash self))
```

To simulate setting a new value for **x**, you could use **(setf x** *value***)**, which would expand into **(send x-hash-table :put-hash self** *value***)**.

15.5 Lambda Macros

Lambda macros are similar to regular Lisp macros, except that regular Lisp macros replace, and expand into, Lisp forms, whereas lambda macros replace, and expand into, Lisp functions. They are an advanced feature, used only for certain special language extensions or embedded programming systems.

To understand what lambda macros do, consider how regular Lisp macros work. When the evaluator is given a Lisp form to evaluate, it inspects the car of the form to figure out what to do. If the car is the name of a function, the function is called. But if the car is the name of a macro, the macro is expanded, and the result of the expansion is considered to be a Lisp form and is evaluated. Lambda macros work analogously, but in a different situation. When the evaluator finds that the car of a form is a list, it looks at the car of this list to figure out what to do. If this car is the symbol **lambda**, the list is an ordinary function, and it is applied to its arguments. But if this car is the name of a lambda macro, the lambda macro is expanded, and the result of the expansion is considered to be a Lisp function and is applied to the arguments.

Like regular macros, lambda macros are named by symbols and have a body, which is a function of one argument. To expand the lambda macro, the evaluator applies this body to the entire lambda macro function (the list whose car is the name of the lambda macro), and expects the body to return another function as its value.

Several special forms are provided for dealing with lambda macros. The primitive for defining a new lambda macro is **lambda-macro**; it is analogous to the **macro** special form. For convenience, **deflambda-macro** and **zl:deflambda-macro-displace** are defined; these work like **defmacro** to provide easy parsing of the function into its component parts. The special form **deffunction** creates a new Lisp function that uses a named lambda macro instead of **lambda** in its definition.

lambda-macro *name lambda-list body...*

Like **macro**, defines a lambda macro to be called *name*. *lambda-list* should be a list of one variable, which is bound to the function being expanded. The lambda macro must return a function. Example:

```
(lambda-macro ilisp (x)
  '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

This defines a lambda macro called **ilisp**. After it has been defined, the following list is a valid Lisp function:

```
(ilisp (x y z) (list x y z))
```

The above function takes three arguments and returns a list of them, but all of the arguments are optional and any extra arguments are ignored. (This shows how to make functions that imitate Interlisp functions, in which all arguments are always optional and extra arguments are always ignored.) So, for example:

```
(funcall #'(ilisp (x y z) (list x y z)) 1 2)  =>  (1 2 nil)
```

deflambda-macro is like **defmacro**, but defines a lambda macro instead of a normal macro.

zl:deflambda-macro-displace is like zl:defmacro-displace, but defines a lambda macro instead of a displacing normal macro.

deffunction defines a function using an arbitrary lambda macro in place of **lambda**. A **deffunction** form is like a **defun** form, except that the function spec is immediately followed by the name of the lambda macro to be used. **deffunction** expands the lambda macro immediately, so the lambda macro must already be defined before **deffunction** is used. For example, suppose the **ilisp** lambda macro were defined as follows:

```
(lambda-macro ilisp (x)
  '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

Then the following example would define a function called **new-list** that would use the lambda macro called **ilisp**:

```
(deffunction new-list ilisp (x y z)
  (list x y z))
```

new-list's arguments are optional, and any extra arguments are ignored. Examples:

```
(new-list 1 2) => (1 2 nil)
(new-list 1 2 3 4) -> (1 2 3)
```

Lambda macro-expander functions can be accessed with the (**:lambda-macro** *name*) function spec.

15.6 Hints to Macro Writers

Over the years, Lisp programmers have discovered useful techniques for writing macros, and have identified pitfalls that must be avoided. This section discusses some of these techniques, and illustrates them with examples.

The most important thing to keep in mind as you learn to write macros is that you should first figure out what the macro form is supposed to expand into, and only then should you start to actually write the code of the macro. If you have a firm grasp of what the generated Lisp program is supposed to look like, from the start, you will find the macro much easier to write.

In general any macro that can be written as an inline function should be written as one, not as a macro, for several reasons:

- Inline functions are easier to write and to read.
- They can be passed as functional arguments (for example, you can pass them to **mapcar**).
- Some subtleties can occur in macro definitions that need not be worried about in inline functions.

See the section "Inline Functions", page 294. A macro can be an inline function only if it has the exact semantics of a function, rather than a special form.

15.6.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write **dolist** as a macro that expanded into a **do**. The first step, as always, is to figure out what the expansion should look like. Let's pick a representative example form, and figure out what its expansion should be. Here is a typical **dolist** form.

```
(dolist (element (append a b))
  (push element *big-list*)
  (foo element 3))
```

We want to create a **do** form that does the thing that the above **dolist** form says to do. That is the basic goal of the macro: it must expand into code that does the same thing that the original code says to do, but it should be in terms of existing Lisp constructs. The **do** form might look like this:

```
(do ((list (append a b) (cdr list))
     (element))
    ((null list))
  (setq element (car list))
  (push element *big-list*)
  (foo element 3))
```

Now we could start writing the macro that would generate this code, and in general convert any **dolist** into a **do**, in an analogous way. However, there is a problem with the above scheme for expanding the **dolist**. The above expansion works fine. But what if the input form had been the following:

```
(dolist (list (append a b))
  (push list *big-list*)
  (foo list 3))
```

This is just like the form we saw above, except that the user happened to decide to name the looping variable **list** rather than **element**. The corresponding expansion would be:

```
(do ((list (append a b) (cdr list))
     (list))
    ((null list))
  (setq list (car list))
  (push list *big-list*)
  (foo list 3))
```

This does not work at all! In fact, this is not even a valid program, since it contains a **do** that uses the same variable in two different iteration clauses.

Here is another example that causes trouble:

```
(let ((list nil))
  (dolist (element (append a b))
    (push element list)
    (foo list 3)))
```

If you work out the expansion of this form, you see that there are two variables named **list**, and that the user meant to refer to the outer one but the generated code for the **push** actually uses the inner one.

The problem here is an accidental name conflict. This can happen in any macro that has to create a new variable. If that variable ever appears in a context in which user code might access it, it might conflict with some other name that is in the user's program.

One way to avoid this problem is to choose a name that is very unlikely to be picked by the user, simply by choosing an unusual name. This will probably work, but it is inelegant since there is no guarantee that the user will not happen to choose the same name. The only sure way to avoid the name conflict is to use an uninterned symbol as the variable in the generated code. The function **(make-symbol "foo")** is useful for creating such symbols.

Here is the expansion of the original form, using an uninterned symbol created by **(make-symbol "foo")**

```
(do ((#:g0005 (append a b) (cdr #:g0005))
     (element))
    ((null #:g0005))
  (setq element (car #:g0005))
  (push element *big-list*)
  (foo element 3))
```

This is the right kind of thing to expand into. Now that we understand how the expansion works, we are ready to actually write the macro. Here it is:

```
(defmacro dolist ((var form) . body)
  (let ((dummy (gensym)))
    `(do ((,dummy ,form (cdr ,dummy))
          (,var))
         ((null ,dummy))
       (setq ,var (car ,dummy))
       . ,body)))
```

Many system macros do not use **(make-symbol "foo")** for the internal variables in their expansions. Instead they use symbols whose print names begin and end with a dot. This provides meaningful names for these variables when looking at the generated code and when looking at the state of a computation in the Debugger. However, this convention means that users should avoid naming variables this way.

15.6.2 prog-Context Conflicts

A related problem occurs when you write a macro that expands into a **prog** (or a **do**, or something that expands into **prog** or **do**) behind the user's back.

Consider the **error-restart** special form; suppose we wanted to implement it as a macro that expands into a **prog**. If it expanded into a standard **prog**, then the following (contrived) Lisp program would not behave correctly:

```
(prog ()
   (setq a 3)
   (error-restart (error "Try again")
     (cond ((> a 10)
            (return 5))
           ((> a 4)
            (fsignal 'lose "You lose."))))
   (setq b 7))
```

The problem is that the **return** returns from the **error-restart** instead of the **prog**. The way to avoid this problem is to use a named **prog** whose name is **t**. The name **t** is special in that it is invisible to the **return** function. If we write **error-restart** as a macro that expands into a **prog** named **t**, then the **return** passes right through the **error-restart** form and returns from the **prog**, as it ought to. Don't use this mechanism if the looping construct needs a **return** from its own operation, since in that case the **return** will exit too far. You might want to use **tagbody** instead.

In general, when a macro expands into a **prog** or a **do** around the user's code, the **prog** or **do** should be named **t** so that **return** forms in the user code return to the right place, unless the macro is documented as generating a **prog/do**-like form that can be exited with **return**.

15.6.3 Macros Expanding Into Many Forms

Sometimes a macro wants to do several different things when its expansion is evaluated. Another way to say this is that sometimes a macro wants to expand into several things, all of which should happen sequentially at run time (not macro-expand time). For example, suppose you wanted to implement **defparameter** as a macro. **defparameter** must do two things: declare the variable to be special, and set the variable to its initial value. To simplify the example, we implement a simplified **defparameter** that does only these two things, and does not have any options. What should a **defparameter** form expand into? What we would like is for an appearance of:

```
(defparameter a (+ 4 b))
```

in a file to be the same thing as the appearance of the following two forms:

```
(proclaim '(special a))
(setq a (+ 4 b))
```

However, because of the way that macros work, they expand into only one form, not two. So we need to have a **defparameter** form expand into one form that is just like having two forms in the file.

There is such a form. It looks like this:

```
(progn
  (proclaim '(special a))
  (setq a (+ 4 b)))
```

In interpreted Lisp, it is easy to see what happens here. This is a **progn** special form, and so all its subforms are evaluated, in turn. The **proclaim** form and the **setq** form are evaluated, and so each of them happens, in turn. So far, so good.

The interesting thing is the way this form is treated by the compiler. The compiler specially recognizes any **progn** form at top level in a file. When it sees such a form, it processes each of the subforms of the **progn** just as if that form had appeared at top level in the file. So the compiler behaves exactly as if it had encountered the **proclaim** form at top level, and then encountered the **setq** form at top level, even though neither of those forms was actually at top level (they were both inside the **progn**). This feature of the compiler is provided specifically for the benefit of macros that want to expand into several things.

Here is the macro definition:

```
(defmacro defparameter (variable init-form)
  `(progn
     (proclaim '(special ,variable))
     (setq ,variable ,init-form)))
```

Here is another example of a form that wants to expand into several things. We will implement a special form called **define-my-command**, which is intended to be used in order to define commands in some interactive user subsystem. For each command, the **define-my-command** form provides two things: a function that executes the command, and a text string that contains the documentation for the command (in order to provide an online interactive documentation feature). This macro is a simplified version of a macro that is actually used in the Zwei editor. Suppose that in this subsystem, commands are always functions of no arguments, documentation strings are placed on the **help** property of the name of the command, and the names of all commands are put onto a list. A typical call to **define-my-command** would look like:

```
(define-my-command move-to-top
  "This command moves you to the top."
  (do ()
      ((at-the-top-p))
    (move-up-one)))
```

This could expand into:

```
(progn
    (defprop
      move-to-top
      "This command moves you to the top."
      help)
    (push 'move-to-top *command-name-list*)
    (defun move-to-top ()
      (do ()
          ((at-the-top-p))
        (move-up-one)))
    )
```

The **define-my-command** expands into three forms. The first one sets up the documentation string and the second one puts the command name onto the list of all command names. The third one is the **defun** that actually defines the function itself. Note that the **defprop** and **push** happen at load time (when the file is loaded); the function, of course, also gets defined at load time. For more discussion of the differences among compile time, load time, and eval time: See the special form **eval-when** in *Program Development Utilities.*

This technique makes Lisp a powerful language in which to implement your own language. When you write a large system in Lisp, frequently you can make things much more convenient and clear by using macros to extend Lisp into a customized language for your application. In the above example, we have created a little language extension: a new special form that defines commands for our system. It lets the writer of the system put documentation strings right next to the code that they document, so that the two can be updated and maintained together. The way that the Lisp environment works, with load-time evaluation able to build data structures, lets the documentation database and the list of commands be constructed automatically.

15.6.4 Macros That Surround Code

There is a particular kind of macro that is very useful for many applications. This is a macro that you place "around" some Lisp code, in order to make the evaluation of that code happen in some context. For a very simple example, we could define a macro called **with-output-in-base**, that executes the forms within its body with any output of numbers that is done defaulting to a specified base.

```
(defmacro with-output-in-base ((base-form) &body body)
   `(let ((base ,base-form))
       ,body))
```

A typical use of this macro might look like:

```
(with-output-in-base (*default-base*)
   (print x)
   (print y))
```

that would expand into:

```
(let ((base *default-base*))
  (print x)
  (print y))
```

This example is too trivial to be very useful; it is intended to demonstrate some stylistic issues. Some special forms are similar to this macro. See the special form **with-open-file** in *Reference Guide to Streams, Files, and I/O*. See the special form **with-input-from-string** in *Reference Guide to Streams, Files, and I/O*. The really interesting thing, of course, is that you can define your own such special forms for your own specialized applications. One very powerful application of this technique is used in a system that manipulates and solves the Rubik's cube puzzle. The system heavily uses a special form called **with-front-and-top**, whose meaning is "evaluate this code in a context in which this specified face of the cube is considered the front face, and this other specified face is considered the top face".

The first thing to keep in mind when you write this sort of macro is that you can make your macro much clearer to people who might read your program if you conform to a set of loose standards of syntactic style. By convention, the names of such special forms start with "**with-**". This seems to be a clear way of expressing the concept that we are setting up a context; the meaning of the special form is "do this *with* the following things true". Another convention is that any "parameters" to the special form should appear in a list that is the first subform of the special form, and that the rest of the subforms should make up a body of forms that are evaluated sequentially with the last one returned. All of the examples cited above work this way. In our **with-output-in-base** example, there was one parameter (the base), which appears as the first (and only) element of a list that is the first subform of the special form. The extra level of parentheses in the printed representation serves to separate the "parameter" forms from the "body" forms so that it is textually apparent which is which; it also provides a convenient way to provide default parameters (a good example is the **with-input-from-string** special form, which takes two required and two optional "parameters"). Another convention/technique is to use the **&body** keyword in the **defmacro** to tell the editor how to correctly indent the special form. See the section "**&**-Keywords Accepted By **defmacro**", page 312.

The other thing to remember is that control can leave the special form either by the last form's returning, or by a nonlocal exit (that is, something doing a **throw**). You should write the special form in such a way that everything is cleaned up appropriately no matter which way control exits. In our **with-output-in-base** example, there is no problem, because nonlocal exits undo lambda-bindings. However, in even slightly more complicated cases, an **unwind-protect** form is needed: the macro must expand into an **unwind-protect** that surrounds the body, with "cleanup" forms that undo the context-setting-up that the macro did. For example, **using-resource** is implemented as a macro that does an **allocate-resource** and then performs the body inside of an **unwind-protect** that has a **deallocate-resource** in its "cleanup" forms. This way the allocated resource item is deallocated whenever control leaves the **using-resource** special form.

15.6.5 Multiple and Out-of-order Evaluation

In any macro, you should always pay attention to the problem of multiple or out-of-order evaluation of user subforms. Here is an example of a macro with such a problem. This macro defines a special form with two subforms. The first is a reference, and the second is a form. The special form is defined to create a cons whose car and cdr are both the value of the second subform, and then to set the reference to be that cons. Here is a possible definition:

```
(defmacro test (reference form)
   `(setf ,reference (cons ,form ,form)))
```

Simple cases work all right:

```
(test foo 3) ==>
  (setf foo (cons 3 3))
```

But a more complex example, in which the subform has side effects, can produce surprising results:

```
(test foo (setq x (1+ x))) ==>
  (setf foo (cons (setq x (1+ x))
                  (setq x (1+ x))))
```

The resulting code evaluates the **setq** form twice, and so **x** is increased by two instead of by one. A better definition of **test** that avoids this problem is:

```
(defmacro test (reference form)
   (let ((value (gensym)))
     `(let ((,value ,form))
         (setf ,reference (cons ,value ,value)))))
```

With this definition, the expansion works as follows:

```
(test foo (setq x (1+ x))) ==>
  (let ((#:g0005 (setq x (1+ x))))
     (setf foo (cons #:g0005 #:g0005)))
```

In general, when you define a new special form that has some forms as its subforms, you have to be careful about when those forms get evaluated. If you are not careful, they can get evaluated more than once, or in an unexpected order, and this can be semantically significant if the forms have side effects. There is nothing fundamentally wrong with multiple or out-of-order evaluation if that is really what you want and if it is what you document your special form to do. However, it is very common for special forms to simply behave like functions, and when they are doing things like what functions do, it is natural to expect them to be function-like in the evaluation of their subforms. Function forms have their subforms evaluated, each only once, in left-to-right order, and special forms that are similar to function forms should try to work that way too for clarity and consistency.

The macro **once-only** makes it easier for you to follow the principle explained above. It is most easily explained by example. The way you would write **test** using **once-only** is as follows:

```
(defmacro test (reference form)
  (once-only (form)
    `(setf ,reference (cons ,form ,form))))
```

This 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. **once-only** automatically introduces a lambda-binding of a generated symbol to hold the value of the form. Actually, it is more clever than that; 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. This is just an optimization that helps produce more efficient code.

The **once-only** macro makes it easier to follow the principle, but it does not completely nor automatically solve the problems of multiple and out-of-order evaluation. It is just a tool that can solve some of the problems some of the time; it is not a panacea.

The following description attempts to explain what **once-only** does, but it is much easier to use **once-only** by imitating the example above than by trying to understand **once-only**'s rather tricky definition.

A **once-only** form looks like:

```
(once-only var-list
  form1
  form2
  ...)
```

var-list is a list of variables. **once-only** is usually used in macros where these variables are Lisp forms. The *forms* are a Lisp program that presumably uses the values of those variables. When the form resulting from the expansion of the **once-only** is evaluated, it first inspects the values of each of the variables in *var-list*; these values are assumed to be Lisp forms. It binds each variable either to its current value, if the current value is a trivial form, or to a generated symbol. Next, **once-only** evaluates the *forms*, in this new binding environment, and when they have been evaluated it undoes the bindings. The result of the evaluation of the last *form* is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables had been bound to trivial forms, then *once-only* just 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 it only evaluates each form once, unless evaluation of the form has no side effects, for each of the forms that were the values of variables in *var-list*. At the same time, no unnecessary lambda-binding appears in this program, but the body of the **once-only** is not cluttered up with extraneous code to decide whether or not to introduce lambda-binding in the program it constructs.

Note well: while **once-only** attempts to prevent multiple evaluation, it does *not* necessarily preserve the *order* of evaluation of the forms! Since it generates the new bindings, the evaluation of complex forms (for which a new variable needs to be created) may be moved ahead of the evaluation of simple forms (such as variable references). **once-only** does not solve all of the problems mentioned in this section.

Caution! A number of system macros, **setf** for example, fail to follow this convention. Unexpected multiple evaluation and out-of-order evaluation can occur with them. This was done for the sake of efficiency and is prominently mentioned in the documentation of these macros. It would be best not to compromise the semantic simplicity of your own macros in this way. (**setf** and related macros follow the convention correctly.)

15.6.6 Nesting Macros

A useful technique for building language extensions is to define programming constructs that employ two special forms, one of which is used inside the body of the other. Here is a simple example. There are two special forms; the outer one is called **with-collection**, and the inner one is called **collect**. **collect** takes one subform, which it evaluates; **with-collection** just has a body, whose forms it evaluates sequentially. **with-collection** returns a list of all of the values that were given to **collect** during the evaluation of the **with-collection**'s body. For example:

```
(with-collection
  (dotimes (i 5)
    (collect i)))

  => (1 2 3 4 5)
```

Remembering the first piece of advice we gave about macros, the next thing to do is to figure out what the expansion looks like. Here is how the above example could expand:

```
(let ((#:g0005 nil))
  (dotimes (i 5)
    (push i #:g0005))
  (nreverse #:g0005))
```

Now, how do we write the definition of the macros? **with-collection** is pretty easy:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       ,@body
       (nreverse ,var))))
```

The hard part is writing **collect**. Let's try it:

```
(defmacro collect (argument)
  `(push ,argument ,var))
```

Note that something unusual is going on here: **collect** is using the variable **var** freely. It is depending on the binding that takes place in the body of **with-collection** to get access to the value of **var**. Unfortunately, that binding took place when **with-collection** got expanded; **with-collection**'s expander function bound **var**, and it got unbound when the expander function was done. By the time the **collect** form gets expanded, **var** has long since been unbound. The macro definitions above do not work. Somehow the expander function of **with-collection** has to communicate with the expander function of **collect** to pass over the generated symbol.

The only way for **with-collection** to convey information to the expander function of **collect** is for it to expand into something that passes that information. What we can do is to define a special variable (which we will call ***collect-variable***), and have **with-collection** expand into a form that binds this variable to the name of the variable that the **collect** should use. Now, consider how this works in the interpreter. The evaluator first sees the **with-collection** form, and calls in the expander function to expand it. The expander function creates the expansion, and returns to the evaluator, which then evaluates the expansion. The expansion includes in it a **let** form to bind ***collect-variable*** to the generated symbol. When the evaluator sees this **let** form during the evaluation of the expansion of the **with-collection** form, it sets up the binding and recursively evaluates the body of the **let**. Now, during the evaluation of the body of the **let**, our special variable is bound, and if the expander function of **collect** gets run, it is able to see the value of **collection-variable** and incorporate the generated symbol into its own expansion.

Writing the macros this way is not quite right. It works fine interpreted, but the problem is that it does not work when we try to compile Lisp code that uses these special forms. When code is being compiled, there is no interpreter to do the binding in our new **let** form; macro expansion is done at compile time, but generated code is not run until the results of the compilation are loaded and run. The way to fix our definitions is to use **compiler-let** instead of **let**. **compiler-let** is a special form that exists specifically to do the sort of thing we are trying to do here. **compiler-let** is identical to **let** as far as the interpreter is concerned, so changing our **let** to a **compiler-let** does not affect the behavior in the interpreter; it continues to work. When the compiler encounters a **compiler-let**, however, it actually performs the bindings that the **compiler-let** specifies, and proceeds to compile the body of the **compiler-let** with all of those bindings in effect. In other words, it acts as the interpreter would.

Here is the right way to write these macros:

```
(defvar *collect-variable*)

(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       (compiler-let ((*collect-variable* ',var))
         . ,body)
       (nreverse ,var))))

(defmacro collect (argument)
  `(push ,argument ,*collect-variable*))
```

A better way to write this type of macro is to use **macrolet** to create a definition of **collect** local to the body of **with-collection**. Example:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       (macrolet ((collect (argument)
                    `(push ,argument ,',var)))
         . ,body)
       (nreverse ,var))))

;To make COLLECT known to editing tools, and to get a better error
;message if it is used in the wrong place, we define a global
;definition that will be shadowed by the MACROLET.  The error
;message for misuse of COLLECT comes out at both compile-time
;and run-time.
```

```
(defmacro collect (argument)
  (compiler:warn () "~S used outside of ~S"
                 'collect 'with-collection)
  '(ferror "~S used outside of ~S"
           '(collect ,,argument) 'with-collection))
```

15.6.7 Functions Used During Expansion

The technique of defining functions to be used during macro expansion deserves explicit mention. A macro-expander function is a Lisp program like any other Lisp program, and it can benefit in all the usual ways by being broken down into a collection of functions that do various parts of its work. Usually macro-expander functions are pretty simple Lisp programs that take things apart and put them together slightly differently, but some macros are quite complex and do a lot of work. Several features of Symbolics Common Lisp, including flavors, **loop**, and **defstruct**, are implemented using very complex macros, which, like any complex, well-written Lisp program, are broken down into modular functions. You should keep this in mind if you ever invent an advanced language extension or ever find yourself writing a five-page expander function.

A particular thing to note is that any functions used by macro-expander functions must be available at compile time. You can make a function available at compile time by surrounding its defining form with **(eval-when (compile load eval) ...)**. Doing this means that at compile time the definition of the function is interpreted, not compiled, and thus runs more slowly. Another approach is to separate macro definitions and the functions they call during expansion into a separate file, often called a "defs" (definitions) file. This file defines all the macros but does not use any of them. It can be separately compiled and loaded up before compiling the main part of the program, which uses the macros. The *system* facility helps keep these various files straight, compiling and loading things in the right order. See the section "Maintaining Large Programs" in *Program Development Utilities.*

15.6.8 Aid for Debugging Macros

The function **mexp** goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder to improve readability). See the section "Formatting Lisp Code" in *Reference Guide to Streams, Files, and I/O.* It terminates when you press the END key. If you type in a form that is not a macro form, there are no expansions and so it does not type anything out, but just prompts you for another form. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

For example:

```
(mexp)
Type End to stop expanding forms

Macro form → (loop named t until nil return 5)
(ZL:LOOP NAMED T UNTIL NI RETURN 5) →
(PROG T NIL
SI:NEXT-LOOP AND NIL
                (GO SI:END-LOOP))
      (RETURN 5)
      (GO SI:NEXT-LOOP)
SI:END-LOOP)

Macro form → (defparameter foo bar) →
(PROGN (EVAL-WHEN (COMPILE)
         (COMPILER:SPECIAL-2 'FOO))
       (EVAL-WHEN (LOAD EVAL)
         (SI:DEFCONST-1 FOO BAR NIL)))
```

See the section "Expanding Lisp Expressions in Zmacs" in *Text Editing and Processing.* That section describes two editor commands that allow you to expand macros – `c-sh-M` and `m-sh-M`. There is also the Command Processor command, Show Expanded Lisp Code. See the document *User's Guide to Symbolics Computers.*

15.7 Displacing Macros

Every time the evaluator sees a macro form, it must call the macro to expand the form. If this expansion always happens the same way, then it is wasteful to expand the whole form every time it is reached; why not just expand it once? A macro is passed the macro form itself, and so it can change the car and cdr of the form to something else by using **rplaca** and **rplacd**! This way the first time the macro is expanded, the expansion is put where the macro form used to be, and the next time that form is seen, it is already expanded. A macro that does this is called a *displacing macro*, since it displaces the macro form with its expansion.

The major problem with this is that the Lisp form gets changed by its evaluation. If you were to write a program that used such a macro, call **grindef** to look at it, then run the program and call **grindef** again, you would see the expanded macro the second time. Presumably the reason the macro is there at all is that it makes the program look nicer; we would like to prevent the unnecessary expansions, but still let **grindef** display the program in its more attractive form. This is done with the function **zl:displace**.

Another thing to worry about with displacing macros is that if you change the definition of a displacing macro, then your new definition does not take effect in any form that has already been displaced. If you redefine a displacing macro, an existing form using the macro uses the new definition only if the form has never been evaluated.

For example **zl:displace**

Replaces the car and cdr of *form* so that it looks like:

```
(si:displaced original-form expansion)
```

form must be a list. *original-form* is equal to *form* but has a different top-level cons so that the replacing mentioned above does not affect it. **si:displaced** is a macro, which returns the caddr of its own macro form. So when the **si:displaced** form is given to the evaluator, it "expands" to *expansion*. **zl:displace** returns *expansion*.

The grinder knows specially about **si:displaced** forms, and grinds such a form as if it had seen the original form instead of the **si:displaced** form.

So if we wanted to rewrite our **addone** macro (See the section "Introduction to Macros", page 285.) as a displacing macro, instead of writing:

```
(macro addone (x)
   (list 'plus '1 (cadr x)))
```

we would write:

```
(macro addone (x)
   (displace x (list 'plus '1 (cadr x))))
```

Of course, we really want to use **defmacro** to define most macros. Since there is no convenient way to get at the original macro form itself from inside the body of a **defmacro**, another version of it is provided:

zl:defmacro-displace is just like **defmacro** except that it defines a displacing macro, using the **zl:displace** function.

Now we can write the displacing version of **addone** as:

```
(defmacro-displace addone (val)
   (list 'plus '1 val))
```

All we have changed in this example is the **defmacro** into **zl:defmacro-displace**. **addone** is now a displacing macro.

15.8 &-Keywords Accepted By defmacro

The pattern in a **defmacro** is like the lambda-list of a normal function. **defmacro** is allowed to contain certain &-keywords.

defmacro destructures all levels of patterns in a consistent way. The inside patterns can also contain **&**-keywords and there is checking of the matching of lengths of the pattern and the subform. See the special form **destructuring-bind** in *Symbolics Common Lisp: Language Dictionary.* This behavior exists for all of **defmacro**'s parameters, except for **&environment**, **&whole**, and **&aux**.

You must use **&optional** in the parameter list if you want to call the macro with less than its full complement of subforms. There must be an exact one-to-one correspondence between the pattern and the data unless you use **&optional** in the parameter destructuring pattern.

```
(defmacro with-output-to-string
        ((var &optional string index) &body body)
  '(let ((with-output-to-string-internal-string
          ,(or string '(make-array 100 :type 'art-string)))
         ...)
     ...
     ,@body))
```

defmacro accepts these keywords:

&optional
: **&optional** is followed by *variable*, (*variable*), (*variable default*), or (*variable default present-p*), exactly the same as in a function. Note that *default* is still a form to be evaluated, even though *variable* is not being bound to the value of a form. *variable* does not have to be a symbol; it can be a pattern. In this case the first form is disallowed because it is syntactically ambiguous. The pattern must be enclosed in a singleton list.

&rest
: Using **&rest** is the same as using a dotted list as the pattern, except that it might be easier to read and leaves a place to put **&aux**.

&key
: Separates the positional parameters and rest parameter from the keyword parameters. See the section "Evaluating a Function Form", page 504.

&allow-other-keys
: In a lambda-list that accepts keyword arguments, **&allow-other-keys** says that keywords that are not specifically listed after **&key** are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.

&aux
: **&aux** is the same in a macro as in a function, and has nothing to do with pattern matching. It separates the destructuring pattern of a macro from the auxiliary variables. Following **&aux** you can put entries of the form:

(variable initial-value-form)

or just *variable* if you want it initialized to **nil** or do not care what the initial value is.

&body	**&body** is identical to **&rest** except that it informs the editor and the grinder that the remaining subforms constitute a "body' rather than "arguments" and should be indented accordingly.
&whole	**&whole** is followed by *variable*, which is bound to the entire macro-call form or subform. *variable* is the value that the macro-expander function receives as its first argument. **&whole** is allowed only in the top-level pattern, not in inside patterns.
&environment	**&environment** is followed by *variable*, which is bound to an object representing the lexical environment where the macro call is to be interpreted. This environment might not be the complete lexical environment. It should be used only with the **macroexpand** function for any local macro definitions that the **macrolet** construct might have established within that lexical environment. **&environment** is allowed only in the top-level pattern, not in inside patterns. See the section "Lexical Environment Objects and Arguments", page 518.

&list-of is not supported as a result of making **defmacro** Common-Lisp compatible. Use **zl:loop** or **mapcar** instead of **&list-of**.

15.9 Functions to Expand Macros

The following functions are provided to allow the user 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 might contain macros.

macroexpand-1

If *form* is a macro form, **macroexpand-1** expands it (once) and returns the expanded form and **t**. Otherwise it returns *form* and **nil**. *env* is a lexical environment that can be supplied to specify the lexical environment of the expansions. See the section "Lexical Environment Objects and Arguments", page 518. *dont-expand-special-forms* prevents macro expansion of forms that are both special forms and macros. See **si:*macroexpand-hook*** below.

macroexpand

If *form* is a macro form, **macroexpand** expands it repeatedly until it is not a macro form and returns two values: the final expansion and **t**. Otherwise, it returns *form* and **nil**. *env* is a lexical environment that can be supplied to specify the lexical environment of the expansions. See the section "Lexical Environment Objects and Arguments", page 518. *dont-expand-special-forms* prevents macro expansion of forms that are both special forms and macros.

si:*macroexpand-hook*

The value of this variable is used as the expansion interface hook by **macroexpand-1**. When **macroexpand-1** determines that a symbol names a macro, it obtains the expansion function for that macro. The value of ***macroexpand-hook*** is called as a function of three arguments: the expansion function, *form*, and *env*. The value returned from this call is the expansion of the macro call.

The initial value of ***macroexpand-hook*** is **funcall**, and the net effect is to invoke the expansion function, giving it *form* and *env* as its two arguments.

PART IV.

Building Programming Constructs

16. Structure Macros

This section contains reference information on the use of **defstruct** and **zl:defstruct**. For an overview of structure macros: See the section "Overview of Structure Macros", page 45.

16.1 Using defstruct And zl:defstruct

defstruct *options* &body *items* *Macro*

defstruct defines a record-structure data type. A call to **defstruct** looks like:

```
(defstruct (name option-1 option-2 ...)
           slot-description-1
           slot-description-2
           ...)
```

name must be a symbol; it is the name of the structure. It is given a **si:defstruct-description** property that describes the attributes and elements of the structure; this is intended to be used by programs that examine other Lisp programs and that want to display the contents of structures in a helpful way. *name* is used for other things; for more information: See the section "Named Structures", page 343.

Because evaluation of a **defstruct** form causes many functions and macros to be defined, you must take care not to define the same name with two different **defstruct** forms. A name can only have one function definition at a time. If a name is redefined, the later definition is the one that takes effect, destroying the earlier definition. (This is the same as the requirement that each **defun** that is intended to define a distinct function must have a distinct name.)

Each *option* can be either a symbol, which should be one of the recognized option names, or a list containing an option name followed by the arguments to the option. Some options have arguments that default; others require that arguments be given explicitly. For more information about options: See the section "Options For **defstruct** And **zl:defstruct**", page 321.

Each *slot-description* can be in any of three forms:

1: *slot-name*
2: *(slot-name default-init)*
3: *((slot-name-1 byte-spec-1 default-init-1)*
(slot-name-2 byte-spec-2 default-init-2)
...)

Each *slot-description* allocates one element of the physical structure, even though in form 3 several slots are defined.

Each *slot-name* must always be a symbol; an accessor function is defined for each slot.

In form 1, *slot-name* simply defines a slot with the given name. An accessor function is defined with the name *slot-name.* The **:conc-name** option allows you to specify a prefix and have it concatenated onto the front of all the slot names to make the names of the accessor functions. Form 2 is similar, but allows a default initialization for the slot. Form 3 lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of **defstruct**.

zl:defstruct *Macro*

zl:defstruct defines a record-structure data type. With Genera 7.0, the **defstruct** macro is available and preferred over **zl:defstruct**. **defstruct** accepts all standard Common Lisp options, and accepts several additional options. **zl:defstruct** is supported for compatibility with previous releases. See the section "Differences Between **defstruct** And **zl:defstruct**", page 334.

The basic syntax of **zl:defstruct** is the same as **defstruct**: See the macro **defstruct**, page 319.

For information on the options that can be given to **zl:defstruct** as well as **defstruct**: See the section "Options For **defstruct** And **zl:defstruct**", page 321.

The **:export** option is accepted by **zl:defstruct** but not by **defstruct**. Stylistically, it is preferable to export any external interfaces in the package declarations instead of scattering **:export** options throughout a program's source files.

:export The **:export** option exports the specified symbols from the package in which the structure is defined. This option accepts the following as arguments: the names of slots and the following options: **:alterant**, **:constructor**, **:copier**, **:predicate**, **:size-macro**, and **:size-symbol**.

The following example shows the use of **:export**.

```
(zl:defstruct (2d-moving-object
               (:type :array)
               :conc-name
               ;; export all accessors and the
               ;; make-2d-moving-object constructor
               (:export :accessors :constructor))
  mass
  x-pos
  y-pos
  x-velocity
  y-velocity)
```

See the section "Importing and Exporting Symbols", page 641.

16.1.1 Options For defstruct And zl:defstruct

This section describes the options that can be given to **defstruct** and **zl:defstruct**. The description of each option states any differences in behavior of the option, when given to **defstruct** and **zl:defstruct**.

The **:export** option can be given to **zl:defstruct** but not **defstruct**. It is described elsewhere: See the macro **zl:defstruct**, page 320.

Here is an example that shows the typical syntax of a call to **defstruct** that gives several options.

```
(cl:defstruct (foo (:type vector)
                   :conc-name
                   (:size-symbol foo))
  a
  b)
```

:type The **:type** option specifies the kind of Lisp object to be used to implement the structure. The option requires one argument, which must be one of the symbols enumerated below, or a user-defined type. If the option itself is not provided, the type defaults to **:array**. You can define your own types by using **defstruct-define-type**.

The **:type** option can be given to both **defstruct** and **zl:defstruct**, but they accept different arguments.

These arguments are accepted by the **defstruct :type** option, but not by the **zl:defstruct :type** option:

vector
Use a vector, storing components as vector elements.

If the structure is **:named**, element 0 of the vector holds the named structure symbol and is therefore not used to hold a component of the structure.

You can use the **:make-array** option with **(:type vector)** to specify the area in which the structures should be made. For example:

```
(defstruct
  (foo (:type vector)
       (:make-array (:area *foo-area*)))
  x y z)
```

(vector *element-type***)**

Use a vector, storing components as vector elements. Each component must be of a type that can be stored in a vector of *element-type*. The structure may be **:named** only if the type **symbol** is a subtype of the specified *element-type*.

If the structure is **:named**, element 0 of the vector holds the named structure symbol and is therefore not used to hold a component of the structure.

These arguments are accepted by the **defstruct :type** option and the **zl:defstruct :type** option:

list Use a list, storing components as list elements.

If the structure is **:named**, the **car** of the list holds the named structure symbol and is therefore not used to hold a component of the structure.

You can use the **:make-list** option with **(:type list)** to specify further options about the list that implements the structure.

:list Same as the **list** option.

:named-list

Like **:list**, but the first element of the list holds the symbol that is the name of the structure and so is not used as a component.

:array

Use an array, storing components in the body of the array.

:named-array

Like **:array**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures", page 343. Element 0 of the array holds the named structure symbol and is therefore not used to hold a component of the structure.

:array-leader
Use an array, storing components in the leader of the array. See the section "Options For **defstruct** And **zl:defstruct**", page 321.

:named-array-leader
Like **:array-leader**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures", page 343. Element 1 of the leader holds the named structure symbol and so is not used to hold a component of the structure.

:tree The structure is implemented out of a binary tree of conses, with the leaves serving as the slots.

:fixnum
This unusual type implements the structure as a single fixnum. The structure must have exactly one slot. This is only useful with the byte field feature; it lets you store a bunch of small numbers within fields of a fixnum, giving the fields names. See the section "Using Byte Fields And **defstruct** Or **zl:defstruct**", page 341.

:grouped-array
See the section "Grouped Arrays", page 343. This option is described there.

:alterant

The **:alterant** option allows you to customize the name of the alterant function. If (**:alterant** *name*) is supplied, the name of the alterant function is *name*. *name* should be a symbol; its print name is the name of the alterant function.

If **:alterant** is specified without an argument, the name of the alterant is **alter-***structure*. This is also the default behavior of **zl:defstruct**, when the **:alterant** option is not given.

If (**:alterant nil**) is specified, no alterant is defined. This is also the default behavior of **defstruct**, when the **:alterant** option is not given.

The following example defines the alterant to be **change-door-slot**.

```
(cl:defstruct (door (:alterant change-door-slot))
  knob-color width)

(setq d (make-door :knob-color 'red :width 5.0))

(change-door-slot d
   knob-color 'blue
   width 5.5)
```

For more information on the use of the alterant macro: See the section "Alterant Macros For **defstruct** And **zl:defstruct** Structures", page 339.

defstruct and zl:defstruct difference

The difference is in the default behavior, when **:alterant** is not supplied.

defstruct	Default is same as **(:alterant nil)**
zl:defstruct	Default is same as **:alterant** without an argument

:but-first The argument to **:but-first** is an accessor from some other structure, and it is expected that this structure will never be found outside that slot of that other structure. Actually, you can use any one-argument function, or a macro that acts like a one-argument function. It is an error for **:but-first** to be used without an argument.

This example should clarify the use of **:but-first**.

```
(cl:defstruct (head (:type list)
                    (:default-pointer person)
                    (:but-first person-head))
  nose
  mouth
  eyes)
```

The **nose** accessor expands like this:

```
(nose x)         ==> (car (person-head x))
(nose)           ==> (car (person-head person))
```

:callable-accessors

This option controls whether accessors are really functions, and therefore "callable", or whether they are macros.

The accessors are functions if this option is not provided,

provided with no argument, or provided with an argument of **t**. Specifically, they are substs, so that they have all the efficiency of macros in compiled programs, while still being function objects that can be manipulated (passed to **mapcar**, and so on).

If this option is provided with an argument of **nil**, then the accessors will be macros, not substs.

Note that if you use the **:default-pointer** option, the accessors cannot be made callable.

:conc-name

The **:conc-name** option allows you to customize the names of the accessor functions. If **(:conc-name** *prefix***)** is supplied, the name of each accessor function is *prefix-slot*. *prefix* should be a symbol; its print name is concatenated onto the front of all the slot names to make the names of the accessor functions.

If **:conc-name** is specified without an argument, the name of each accessor is *structure-slot*; that is, the name of the structure followed by a hyphen, followed by the slot name. This is also the default behavior of **defstruct**, when the **:conc-name** option is not given.

:conc-name changes the name of the accessor functions, but has no effect on slot names that are given to the constructor and alterant macros. Thus when you use **:conc-name**, the slot names and accessor names are different.

In the following example, the **:conc-name** option specifies the prefix **"get-door-"**, which causes the accessor functions to be named **get-door-knob-color** and **get-door-width**.

```
(cl:defstruct (door (:conc-name get-door-))
  knob-color
  width)

(setq d (make-door :knob-color 'red :width 5.0))

(get-door-knob-color d) ==> red
```

If **(:conc-name nil)** is specified, the name of each accessor is *slot*, the name of the slot. This is also the default behavior of **zl:defstruct**, when the **:conc-name** option is not given. When the name of the accessor is just *slot*, you should name the slots according to a suitable convention. You should always prefix the names of all accessor functions with some text unique to the structure.

defstruct and zl:defstruct difference

The difference is in the default behavior, when **:conc-name** is not supplied.

defstruct	Default is same as **:conc-name** without an argument.
zl:defstruct	Default is same as (**:conc-name nil**).

:constructor

This option is accepted by both **defstruct** and **zl:defstruct**.

This option takes one argument, which specifies the name of the constructor. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string "**make-**" to the name of the structure. If the argument is provided and is **nil**, no constructor is defined. A more general form of this option is also available: See the section "By-position Constructors For **defstruct** And **zl:defstruct** Structures", page 339.

For more information about the use of the constructor: See the section "Constructors For **defstruct** And **zl:defstruct** Structures", page 337.

defstruct and zl:defstruct difference

defstruct	Defines a constructor function.
zl:defstruct	Defines a constructor macro.

:copier

The **:copier** option allows you to customize the name of the copier function. If (**:copier** *name*) is supplied, the name of the copier function is *name*. *name* should be a symbol; its print name is the name of the copier function.

If **:copier** is specified without an argument, the name of the copier function is **copy-***structure*. This is also the default behavior of **defstruct**, when the **:copier** option is not given.

If (**:copier nil**) is specified, no copier is defined. This is also the default behavior of **zl:defstruct**, when the **:copier** option is not given.

For example:

```
(cl:defstruct (foo (:type list) :copier)
  foo-a
  foo-b)
```

This example would generate a function named **copy-foo**, with a definition approximately like this:

```
(defun copy-foo (x)
  (list (car x) (cadr x)))
```

defstruct and zl:defstruct difference

The difference is in the default behavior, when **:copier** is not supplied.

defstruct	Default is same as **:copier** without an argument
zl:defstruct	Default is same as **(:copier nil)**

:default-pointer Normally, the accessors defined by **defstruct** expect to be given exactly one argument. However, if the **:default-pointer** argument is used, the argument to each accessor is optional. You can continue to use the accessor function in the usual way. You can also invoke an accessor it without its argument; it behaves as if you had invoked it on the result of evaluating the form that is the argument to the **:default-pointer** argument. For example:

```
(cl:defstruct (room (:default-pointer *room-13*)
                    :conc-name)
  name
  contents)


(setq play-room
      (make-room :name 'den :contents 'tv))
(setq *room-13*
      (make-room :name 'kitchen :contents 'fridge))


(room-name play-room) ==> DEN
(room-name) ==> KITCHEN
```

If the argument to the **:default-pointer** argument is not given, it defaults to the name of the structure.

:eval-when Normally the functions and macros defined by **defstruct** are defined at eval time, compile time, and load time. This option allows you to control this behavior. The argument to the

:eval-when option is just like the list that is the first subform of an **eval-when** special form. For example: **(:eval-when (:eval :compile))** causes the functions and macros to be defined only when the code is running interpreted or inside the compiler.

:include This option is used for building a new structure definition as an extension of an old structure definition. Suppose you have a structure called **person** that looks like this:

```
(defstruct (person :conc-name)
  name
  age
  sex)
```

Now suppose you want to make a new structure to represent an astronaut. Since astronauts are people too, you would like them to also have the attributes of name, age, and sex, and you would like Lisp functions that operate on **person** structures to operate just as well on **astronaut** structures. You can do this by defining **astronaut** with the **:include** option, as follows:

```
(defstruct (astronaut (:include person))
  helmet-size
  (favorite-beverage 'tang))
```

The **:include** option inserts the slots of the included structure at the front of the list of slots for this structure. That is, an **astronaut** will have five slots; first the three defined in **person**, then the two defined in **astronaut** itself. The accessor functions defined by the **person** structure can be applied to instances of the **astronaut** structure. The following illustrates how you can use **astronaut** structures:

```
(setq x (make-astronaut name 'buzz
                        age 45
                        sex t
                        helmet-size 17.5))

(person-name x) => buzz
(favorite-beverage x) => tang
```

Note that the **:conc-name** option was *not* inherited from the included structure; it applies only to the accessor functions of **person** and not to those of **astronaut**. Similarly, the **:default-pointer** and **:but-first** options, as well as the **:conc-name** option, apply only to the accessor functions for the structure in which they are enclosed; they are not inherited if you include a structure that uses them.

The argument to the **:include** option is required, and must be the name of some previously defined structure of the same type as this structure. **:include** does not work with structures of type **:tree** or of type **:grouped-array**.

The following is an advanced feature. Sometimes, when one structure includes another, the default values for the slots that came from the included structure are not what you want. The new structure can specify different default values for the included slots than the included structure specifies, by giving the **:include** option as:

(:include *name new-init-1 ... new-init-n*)

Each *new-init* is either the name of an included slot or a list of the form (*name-of-included-slot init-form*). If it is just a slot name, the slot has no initial value in the new structure. Otherwise its initial value form is replaced by the *init-form*. The old (included) structure is unmodified.

For example, to define **astronaut** so that the default age for an astronaut is **45**, then the following can be used:

```
(defstruct (astronaut (:include person (age 45)))
   helmet-size
   (favorite-beverage 'tang))
```

:initial-offset This allows you to tell **defstruct** to skip over a certain number of slots before it starts allocating the slots described in the body. This option requires an argument (which must be a fixnum) that is the number of slots you want **defstruct** to skip. To use this option, you must understand how **defstruct** is implementing your structure; otherwise, you will be unable to make use of the slots that **defstruct** has left unused.

:make-list You can use the **:make-list** option with **(:type list)** or **(:type :list)** to specify further options about the list that implements the structure. For example, you can specify the area in which the structures should be made.

```
(defstruct
  (foo (:type list)
       (:make-list (:area *foo-area*)))
  x y z)
```

:make-array If the structure being defined is implemented as an array, this option can be used to control those aspects of the array that are 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 might want to specify the type of the array. Of course, this option is only meaningful if the structure is, in fact, being implemented by an array.

The argument to the **:make-array** option should be a list of alternating keyword symbols to the **make-array** function, and forms whose values are the arguments to those keywords. For example, **(:make-array (:area *foo-area*))** requests that structures of this type be consed in ***foo-area***. Note that the keyword symbol is not evaluated.

When necessary, **defstruct** overrides any of the **:make-array** options. For example, if your structure is of type **:array**, then **defstruct** supplies the size of that array, regardless of what you say in the **:make-array** option.

Constructor macros for structures implemented as arrays all allow the keyword **:make-array** to be supplied. Attributes supplied therein override any **:make-array** option attributes supplied in the original **defstruct** form. If some 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 lets you control the initialization of arrays created by **defstruct** as instances of structures. **make-array** initializes the array before the constructor code does. Therefore, any initial value supplied via the new **:initial-value** keyword for **make-array** is overwritten in any slots where you gave **defstruct** an explicit initialization.

If a structure is of type **:array-leader**, you probably want to specify the dimensions of the array. The dimensions of an array are given to **:make-array** as a position argument rather than a keyword argument, so there is no way to specify them in the above syntax. To solve this problem, you can use the keyword **:dimensions** or the keyword **:length** (they mean the same thing) with a value that is anything acceptable as **make-array**'s first argument.

:named This means that you want to use one of the "named" types. If you specify a type of **:array**, **:array-leader**, or **:list**, and give the **:named** option, then the **:named-array**, **:named-array-leader**, or **:named-list** type is used instead. Asking for type **:array** and giving the **:named** option as well is the same as asking for the type **:named-array**; the only difference is stylistic.

:predicate The **:predicate** option allows you to customize the name of the predicate function. The predicate function recognizes objects of this structure. If **(:predicate** *name***)** is supplied, the name of the predicate function is *name*. *name* should be a symbol; its print name is the name of the predicate function. The **:predicate** option works only for named types.

If **:predicate** is specified without an argument, the name of the predicate is *structure*-**p**. This is also the default behavior of **defstruct**, when the **:predicate** option is not given.

If **(:predicate nil)** is specified, no predicate is defined. This is also the default behavior of **zl:defstruct**, when the **:predicate** option is not given.

The following example defines a single-argument predicate function, **foo-p**, that returns **t** only for objects of structure **foo**.

```
(cl:defstruct (foo :named :predicate)
  foo-a
  foo-b)
```

The following example defines a predicate function called **is-it-a-foo?**.

```
(cl:defstruct (foo :named (:predicate is-it-a-foo?))
  foo-a
  foo-b)
```

defstruct and zl:defstruct difference

The difference is in the default behavior, when **:predicate** is not supplied.

defstruct	If **:type** option is not given, or if both **:type** and **:named** are given, default is same as **:predicate** without an argument. If **:type** option is given and **:named** is not given, default is same as **(:predicate nil)**.
zl:defstruct	Default is same as **(:predicate nil)** regardless of whether the **:type** option is given.

:print

The Common Lisp **:print-function** option has the same effect as this option.

The **:print** option gives you implementation-independent control over the printed representation of a structure. Using this option defeats the **sys:printing-random-object** mechanism. See the macro **sys:printing-random-object** in *Reference Guide to Streams, Files, and I/O.*

The **:print** option takes a format string and its arguments. The arguments are evaluated in an environment in which the name symbol for the structure is bound to the structure instance being printed.

The **:print** option makes obsolete the use of a **named-structure-invoke** handler to define **:print** handlers.

:print-function

The **:print-function** option is accepted by **defstruct** but not by **zl:defstruct**. The argument is a function to be used to print this type of structure. The printer uses the print function for structures of unspecified type and when the type is explicitly specified as a named vector. The printer never uses a print function for a structure implemented as a named list, but the **describe-defstruct** function does.

The print function should accept three arguments: the structure to be printed, the stream, and an integer indicating the current depth. The function must be acceptable to the **function** special form.

The function must respect the following print control variables: ***print-escape***, ***print-pretty***, and ***print-structure-contents***.

You can use the function **sys:print-cl-structure** or the macro **sys:print-cl-structure** in a printer function. See the function **sys:print-cl-structure** in *Symbolics Common Lisp: Language Dictionary.* See the macro **sys:cl-structure-printer** in *Symbolics Common Lisp: Language Dictionary.*

```
(defun file-branch-print-function (b stream depth)
  (if *print-escape*
    (if *print-structure-contents*
       (sys:cl-structure-printer file-branch b stream depth)
       (sys:printing-random-object (b stream :typep)
         (format stream "~A" (file-branch-name b))))
    (format stream "~A" (file-branch-name b))))
```

Common Lisp specifies that **:print-function** may be used only if **:type** is not used; however, Genera does not enforce this restriction.

: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 instantiation of the structure.) The **:property** option can be used to give a **defstruct** an arbitrary property. (**:property** *property-name value*) gives the **defstruct** a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user should look inside the **si:defstruct-description** structure. See the section "**defstruct** And **zl:defstruct** Internal Structures", page 351.
:size-symbol	The **:size-symbol** option allows you to specify a global variable whose value is the "size" of the structure; this variable is declared with **zl:defconst**. The exact meaning of the size varies, but in general this number is the one you would need to know if you were going to allocate one of these structures yourself. The symbol has this value both at compile time and at run time. If this option is present without an argument, then the name of the structure is concatenated with "**-size**" to produce the symbol.
:size-macro	This is similar to the **:size-symbol** option. A macro of no arguments is defined that expands into the size of the structure. The name of this macro defaults as with **:size-symbol**.
:times	This option is used for structures of type **:grouped-array** to control the number of repetitions of the structure that are allocated by the constructor macro. The constructor macro also allows **:times** to be used as a keyword that overrides the value given in the original **defstruct** form. If **:times** appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor allocates only one instance of the structure.
type	In addition to the documented options to **defstruct** and **zl:defstruct**, any currently defined type (any valid argument to the **:type** option) can be used as an option. This is mostly for compatibility with older versions of **zl:defstruct**. It allows you to say just *type* instead of (**:type** *type*). It is an error to give an argument to one of these options.
other	Finally, if an option is not found among the other options,

defstruct or **zl:defstruct** checks the property list of the name of the option to see if it has a non-**nil** **:defstruct-option** property. If it does have such a property, then if the option was of the form (*option-name value*), it is treated just like (**:property** *option-name value*). That is, the structure is given an *option-name* property of *value*. It is an error to use such an option without a value.

This provides a primitive way for you to define your own options to **defstruct** or **zl:defstruct**, particularly in connection with user-defined types. See the section "Extensions To **defstruct** And **zl:defstruct**", page 346. Several options to **defstruct** and **zl:defstruct** are implemented using this mechanism.

16.1.2 Differences Between defstruct And zl:defstruct

defstruct and **zl:defstruct** provide a similar functionality. **defstruct** adheres to the Common Lisp standard, with several extensions that were derived from useful features of **zl:defstruct**. **zl:defstruct** is supported for compatibility with previous releases.

Most of the documentation on **defstruct** pertains equally well to **zl:defstruct**. (See the section "Structure Macros", page 319.) This section describes the differences between **defstruct** and **zl:defstruct**.

- The constructor

  defstruct defines constructor functions, but **zl:defstruct** defines constructor macros.

  When using the constructor macro for a **zl:defstruct**-defined structure, you give the names of the slots as the arguments to initialize the slots.

  ```
  (zl:defstruct door1 knob-color)
  (make-door1 knob-color 'blue)    ;slot name alone
  ```

  When using the constructor function for a **defstruct**-defined structure, you give keyword arguments with the same name as the slots, to initialize the slots.

  ```
  (cl:defstruct door2 knob-color)
  (make-cl-door2 :knob-color 'red)  ;slot name in keyword package
  ```

- Options to **defstruct** and **zl:defstruct**

  Most of the options accepted by **defstruct** are also accepted by **zl:defstruct**. Some of the options that are accepted by both have a slightly different behavior when given to **defstruct** than when given to **zl:defstruct**. The option with the most notable differences is **:type**. These differences are explicitly stated in the documentation: See the section "Options For **defstruct** And **zl:defstruct**", page 321.

The **:print-function** and **:constructor-make-array-keywords** options are accepted by **defstruct** but not by **zl:defstruct**. The **:export** option is accepted by **zl:defstruct** but not by **defstruct**.

• Default behavior for **defstruct** and **zl:defstruct**

defstruct and **zl:defstruct** behave differently when no options are given. The differences in default behavior are noted below.

defstruct Default Behavior:

° The structure is implemented as a named vector. This means that by default, the **:named** option is implied. However, if you supply the **:type** option, the **:named** option is no longer implied; you should specify **:named** explicitly if you want a named structure.

° The name of the structure becomes a valid type specifier for **typep**.

° Accessor functions are defined for each slot, named by the convention:

structure-slot

° No alterant is defined, but you can use **setf** with an accessor function to change a slot value, such as:

`(setf` (*accessor object*) *new-value*)

° A copier function is defined, named by the convention:

copy-*structure*

° If the **:type** option is not given, or the **:type** and **:named** options are both given, a predicate function is defined, named by the convention:

structure-**p**

However, if **:type** is given and **:named** is not given, no predicate function is defined.

zl:defstruct Default Behavior:

° The structure is implemented as an unnamed array.

- ° The name of the structure does not become a valid type specifier for **typep**.
- ° Accessor functions are defined for each slot, named by the convention:

  slot

- ° An alterant function is defined, named by the convention:

  alter-*structure*

  You can also use **setf** with an accessor function to change a slot value.

  `(setf` (*accessor object*) *new-value*)

- ° No copier function is defined.
- ° No predicate function is defined.

16.2 defstruct And zl:defstruct Structures And type-of

Under certain circumstances, **defstruct** and **zl:defstruct** define the name of the structure as a type name in both the Common Lisp and Zetalisp type systems. In these circumstances it is illegal for the name of the structure to be the same as the name of an existing type (including a flavor or a built-in type).

The name of the structure is defined as a type name when the structure is defined in one of these ways:

- With **defstruct**, when the **:type** option is not given
- With **defstruct**, when the **(:type :vector)** and **:named** options are given
- With **defstruct**, when the **(:type (:vector** *element***))** and **:named** options are given
- With **zl:defstruct**, when the **(:type :named-array)** option is given
- With **zl:defstruct**, when the **(:type :array)** and **:named** options are given
- With **zl:defstruct**, when the **(:type :named-array-leader)** option is given
- With **zl:defstruct**, when the **(:type :array-leader)** and **:named** options are given

When a structure is defined as a type name, **(type-of** *object***)** returns the symbol that is the name of the object's structure.

(typep *object 'structure-name***)** and **(zl:typep** *object 'structure-name***)** return **t** if the flavor of *object* is named *structure-name*, **nil** otherwise.

16.3 Using the Constructor and Alterant Macros For defstruct And zl:defstruct Structures

The documentation in this section regarding **defstruct** also applies to **zl:defstruct**.

This section describes how to create instances of structures and alter the values of its slots. After you have defined a new structure with **defstruct**, you can create instances of this structure using the constructor, and you can alter the values of its slots using the alterant macro.

By default, **defstruct** defines a constructor function, forming its name by concatenating "**make-**" onto the name of the structure. If you use the **:alterant** option with no argument, an alterant macro is defined, its name formed by concatenating "**alter-**" onto the name of the structure.

You can specify the names of the constructor or alterant macros by passing the name you want to use as the argument to the **:constructor** or **:alterant** options. You can also specify that you do not want the macro created at all by passing **nil** as the argument.

16.3.1 Constructors For defstruct And zl:defstruct Structures

Note that **defstruct** implements the constructor as a function, but **zl:defstruct** implements it as a macro.

A call to a constructor has the form:

```
(name-of-constructor
        symbol-1 form-1
        symbol-2 form-2
        ...)
```

Each *symbol* indicates a slot of the structure (this is not necessarily the same as the name of the accessor). *symbol* can also be one of the specially recognized keywords described further on. If *symbol* indicates a *slot*, that element of the created structure is initialized to the value of the corresponding *form*. All the *forms* are evaluated.

When using the constructor for a **defstruct**-defined structure, a *symbol* that indicates a slot is the name of the slot in the keyword package.

```
(cl:defstruct door1
   knob-color
   width)

(make-door1 :knob-color 'red    ;slot name in keyword package
            :width 5.5)
```

When using the constructor for a **zl:defstruct**-defined structure, a *symbol* that indicates a slot is just the name of the slot.

```
(zl:defstruct door2
   knob-color
   width)

(make-door2 knob-color 'red    ;slot name
            width 5.5)
```

If no *symbol* is present for a given slot, then the slot is initialized to the result of evaluating the default initialization form specified in the call to **defstruct**. In other words, the initialization form specified to the constructor overrides the initialization form specified to **defstruct**. If the **defstruct** itself also did not specify any initialization, the element's initial value is undefined.

Two symbols are specially recognized by the constructor:

:make-array	Should be used only for **:array** and **:array-leader** type structures, or the named versions of those types
:times	Should be used only for **:grouped-array** type structures.

If one of these symbols appears instead of a slot name, then it is interpreted just as the **:make-array** option or the **:times** option, and it overrides what was requested in that option.

For example:

```
(make-ship ship-x-position 10.0
           ship-y-position 12.0
           :make-array (:leader-length 5 :area disaster-area))
```

The order of evaluation of the initialization forms is not necessarily the same as the order in which they appear in the constructor call, nor the order in which they appear in the **defstruct**. You should make sure your code does not depend on the order of evaluation.

The *forms* are reevaluated every time a constructor is called. For example, if the form **(gensym)** is used as an initialization form (either in a call to a constructor or as a default initialization in the **defstruct**) then every call to the constructor would create a new symbol.

16.3.2 By-position Constructors For defstruct And zl:defstruct Structures

Note that **defstruct** defines a constructor function, but **zl:defstruct** defines a constructor macro.

If the **:constructor** option is given as (**:constructor** *name arglist*), then instead of making a keyword-driven constructor, **defstruct** or **zl:defstruct** defines a constructor that takes arguments whose meaning is determined by the argument's position rather than by a keyword. The *arglist* is used to describe what the arguments to the constructor will be. In the simplest case something like **(:constructor make-foo (a b c))** defines **make-foo** to be a three-argument constructor whose arguments are used to initialize the slots named **a**, **b**, and **c**.

In addition, you can use the keywords &optional, &rest, and &aux in the argument list. They work as you might expect, but note the following:

```
(:constructor make-foo
        (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines **make-foo** to be a constructor of one or more arguments. The first argument is used to initialize the **a** slot. The second argument is used to initialize the **b** slot. If there is no second argument, then the default value (if any) given in the body of the **defstruct** or **zl:defstruct** is used instead. The third argument is used to initialize the **c** slot. If there is no third argument, then the symbol **sea** is used instead. Any arguments following the third argument are collected into a list and used to initialize the **d** slot. If there are three or fewer arguments, then **nil** is placed in the **d** slot. The **e** slot *is not initialized*; its initial value is undefined. Finally, the **f** slot is initialized to contain the symbol **eff**.

The actions taken in the **b** and **e** cases were carefully chosen to allow you to specify all possible behaviors. Note that the &aux "variables" can be used to completely override the default initializations given in the body.

Note that you are allowed to give the **:constructor** option more than once, so that you can define several different constructors, each with a different syntax.

The following restrictions should also be noted:

- For **zl:defstruct** these "function-style" constructors do not guarantee that their arguments will be evaluated in the order that you wrote them.
- You cannot specify the **:make-array** or **:times** information in this form of constructor.

16.3.3 Alterant Macros For defstruct And zl:defstruct Structures

A call to the alterant macro has the form:

(*name-of-alterant-macro object*
    slot-name-1 form-1
    slot-name-2 form-2
    ...)

object is evaluated, and should return an object of the structure. Each *form* is evaluated, and the corresponding slot is changed to have the result as its new value. The slots are altered after all the *forms* are evaluated, so you can exchange the values of two slots, as follows:

```
(alter-ship enterprise
    ship-x-position (ship-y-position enterprise)
    ship-y-position (ship-x-position enterprise))
```

As with the constructor macro, the order of evaluation of the *forms* is undefined. Using the alterant macro can produce more efficient code than using consecutive **setfs** when you are altering two byte fields of the same object, or when you are using the **:but-first** option.

You can use alterant macros on structures whose accessors require additional arguments. Put the additional arguments before the list of slots and values, in the same order as required by the accessors.

16.4 Functions Related To defstruct Structures

This summary briefly describes the functions related to **defstruct** structures.

defstruct	Defines a new aggregate data structure with named components.
zl:defstruct	Defines a new aggregate data structure with named components.
describe-defstruct	Prints out a description of a give instance of a structure, including the contents of each of its slots.
defstruct-define-type	Teaches **defstruct** and **zl:defstruct** about new types that it can use to implement structures.
sys:print-cl-structure	Function intended for use in a **defstruct :print-function** option; enables you to respect ***print-escape***.
sys:cl-structure-printer	Macro intended for use in a **defstruct :print-function** option; enables you to respect ***print-escape***.

16.5 Using Byte Fields And defstruct Or zl:defstruct

The byte field feature of **defstruct** or **zl:defstruct** allows you to specify that several slots of your structure are bytes in an integer stored in one element of the structure. For example, consider the following structure:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  (area-code 617)
  exchange
  line-number)
```

Although this works correctly, it wastes space. Area codes and exchange numbers are always less than **1000**, and so both can fit into **10** bit fields when expressed as binary numbers. To tell **defstruct** or **zl:defstruct** to do so, you can change the structure definition to one of the following forms.

Using **defstruct** the syntax is:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  ((line-number)
   (area-code 617 :byte (byte 10 10))
   (exchange 0 :byte (byte 10 0))))
```

Using **zl:defstruct** the syntax is:

```
(zl:defstruct (phone-book-entry (:type :list))
   name
   address
   ((area-code (byte 10 10) 617)
    (exchange (byte 10 0))
    (line-number)))
```

The lists **(byte 10 10)** and **(byte 10 0)** are byte specifiers to be used with the functions **ldb** and **dpb**. The accessors, constructor, and alterant macros now operate as follows:

```
(setq pbe (make-phone-book-entry
            :name "Fred Derf"
            :address "259 Orchard St."
            :exchange 232
            :line-number 7788))
```

```
==> (list "Fred Derf" "259 Orchard St." (dpb 232 12 2322000) 17154)

(phone-book-entry-area-code pbe) ==> (LDB (BYTE 10 10) (NTH 2 FOO))

(alter-phone-book-entry pbe
   area-code ac
   exchange ex)

==> ((lambda (g0530)
       (setf (nth 2 g0530)
             (dpb ac 1212 (dpb ex 12 (nth 2 g0530)))))
     pbe)
```

Note that the alterant macro is optimized to read and write the second element of the list only once, even though you are altering two different byte fields within it. This is more efficient than using two **setfs**. Additional optimization by the alterant macro occurs if the byte specifiers in the **defstruct** slot descriptions are constants.

If the byte specifier is **nil**, the accessor is defined to be the usual kind that accesses the entire Lisp object, thus returning all the byte field components as a fixnum. These slots can have default initialization forms.

The byte specifier need not be a constant; you can use a variable (or any Lisp form). It is evaluated each time the slot is accessed. Of course, you do not ordinarily want the byte specifier to change between accesses.

Constructor 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** or **zl:defstruct** form.

2. Initializations for the byte fields given in the **defstruct** or **zl:defstruct** form.

3. Initializations for the entire word given in the constructor macro form.

4. Initializations for the byte fields given in the constructor macro form.

Alterant macros work similarly: the modification for the entire Lisp object is done first, followed by modifications to specific byte fields. If any byte fields being initialized or altered overlap each other, the actions of the constructor and alterant macros are unpredictable.

16.6 Grouped Arrays

The grouped array feature allows you to store several instances of a structure side-by-side within an array. This feature is somewhat limited; it does not support the **:include** and **:named** options.

The accessor functions are defined to take an extra argument, which should be an integer, and is the index into the array of where 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 the structure is the second argument, the opposite of what you might expect. This is because the structure is &optional if the **:default-pointer** option is used.

Note also that the "size" of the structure (for purposes of the **:size-symbol** and **:size-macro** options) 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 macro is given as the argument to the **:times** option to **defstruct** or **zl:defstruct**, or the **:times** keyword of the constructor macro.

16.7 Named Structures

16.7.1 Introduction to Named Structures

The *named structure* feature provides a very simple form of user-defined data type. Any array can be made a named structure: See the function **zl:make-array-into-named-structure** in *Symbolics Common Lisp: Language Dictionary*.

However, usually the **:named** option of **defstruct** is used to create named structures. See the section "**defstruct** And **zl:defstruct** Structures And **type-of**", page 336.

The principal advantages of a named structure are that it has a more informative printed representation than a normal array and that the **describe** function knows how to give a detailed description of it. (You do not have to use **describe-defstruct**, because **describe** can figure out the names of the structure's slots by looking at the named structure's name.) It is recommended, therefore, that "system" data structures be implemented with named structures.

Flavors offers another kind of user-defined data type, more advanced but less efficient when used only as a record structure: See the section "Flavors", page 353.

A named structure has an associated symbol called its "named structure symbol"; it represents the user-defined type of which it is an instance. The **type-of** function applied to the named structure returns this symbol. If the array has a leader, the symbol is found in element 1 of the leader; otherwise it is found in element 0 of the array. Note: If a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be stored in any element of a numeric array.

If you call **typep** with two arguments, the first an instance of a named structure and the second its named structure symbol, it returns **t**. **t** is also returned if the second argument is the named structure symbol of a **:named defstruct** included (using the **:include** option), directly or indirectly, by the **defstruct** for this structure. For example, if the structure **astronaut** includes the structure **person**, and **person** is a named structure, then giving **typep** an instance of an **astronaut** as the first argument, and the symbol **person** as the second argument, returns t. This reflects the fact that an astronaut is, in fact, a person, as well as an astronaut.

16.7.2 Handler Functions for Named Structures

You can associate a function that handles various operations that can be done on the named structure with a named structure. You can control both how the named structure is printed and what **describe** will do with it.

To provide such a handler function, make the function the **named-structure-invoke** property of the named structure symbol. The functions that know about named structures apply this handler function to several arguments. The first is a "keyword" symbol to identify the calling function, and the second is the named structure itself. The rest of the arguments passed depend on the caller; any named structure function should have a "&rest" parameter to absorb any extra arguments that might be passed. What the function is expected to do depends on the keyword it is passed as its first argument. The following keywords are defined:

:which-operations
Returns a list of the names of the operations handled by the function.

:print-self
The arguments are **:print-self**, the named structure, the stream to which to output, the current depth in list-structure, and **t** if slashification is enabled (**prinl** versus **princ**). The printed representation of the named structure should be output to the stream. If the named structure symbol is not defined as a function, or **:print-self** is not in its **:which-operations** list, the printer defaults to a reasonable printed representation. For example:

#<named-structure-symbol octal-address>

:describe The arguments are **:describe** and the named structure. It should output a description of itself to ***standard-output***. If the named structure symbol is not defined as a function, or **:describe** is not in its **:which-operations** list, the describe system checks whether the named structure was created by using the **:named** option of **defstruct**; if so, the names and values of the structure's fields are enumerated.

Here is an example of a simple named-structure handler function. For this example to have any effect, the person **defstruct** used as an example there must be modified to include the **:named** attribute.

```
(defselect ((:property person named-structure-invoke))
           (:print-self (person stream ignore slashify-p)
            (format stream
                    (if slashify-p "#<person ~a>" "~a")
           (person-name person))))
```

This example causes a person structure to include its name in its printed representation; it also causes **princ** of a person to print just the name, with no "#<" syntax.

Even though the astronaut structure there **:includes** the person structure, this named-structure handler is not invoked when an astronaut is printed, and an astronaut does not include his name in his printed representation. This is because named structures are not as general as flavors.

In this example, the **:which-operations** handler is automatically generated, as well as the handlers for **:operation-handled-p** and **:send-if-handles**.

Another way to write this handler is as follows:

```
(defselect ((:property person named-structure-invoke))
  (:print-self (person stream ignore slashify-p)
   (if slashify-p
       (si:printing-random-object (person stream :typep)
         (princ (person-name person) stream))
       (princ (person-name person) stream))))
```

This example uses the **sys:printing-random-object** special form, which is a more advanced way of printing #< **...** >. It interacts with the **si:print-readably** variable and special form.

16.7.3 Functions That Operate on Named Structures

named-structure-p
Returns **nil** if the given object is not a named structure.

named-structure-symbol
Returns the named structure symbol of the given named structure.

zl:make-array-into-named-structure
Turns the given array into a named structure.

named-structure-invoke
Calls the handler function of the named structure symbol.

Also refer to the **:named-structure-symbol** keyword to **make-array**.

16.8 Extensions To defstruct And zl:defstruct

This section describes the use of **defstruct-define-type**.

16.8.1 An Example Of defstruct-define-type

This section provides an explanation of how **defstruct-define-type** works by examining a call to the macro. This is how the **:list** type of structure might have been defined:

```
(defstruct-define-type :list
        (:cons (initialization-list description keyword-options)
               :list
               '(list . ,initialization-list))
        (:ref (slot-number description argument)
              '(nth ,slot-number ,argument)))
```

This is the simplest possible form of **defstruct-define-type**. It provides **defstruct** with two Lisp forms: one for creating forms to construct instances of the structure, and one for creating forms to become the bodies of accessors for slots of the structure.

The keyword **:cons** is followed by a list of three variables that are bound while the constructor-creating form is evaluated. The first, **initialization-list**, is bound to a list of the initialization forms for the slots of the structure. The second, **description**, is bound to the **si:defstruct-description** structure for the structure. See the section "**defstruct** And **zl:defstruct** Internal Structures", page 351. For a description of the third variable, **keyword-options**, and the **:list** keyword: See the section "Options To **defstruct-define-type**", page 347.

The keyword **:ref** is followed by a list of three variables that are bound while the accessor-creating form is evaluated. The first, **slot-number**, is bound to the number of the slot that the new accessor should reference. The second, **description**, is bound to the **si:defstruct-description** structure for the structure. The third, **argument**, is bound to the form that was provided as the argument to the accessor.

defstruct-define-type *type* &body *options* *Macro*

Teaches **defstruct** and **zl:defstruct** about new types that it can use to implement structures.

The body of this function is shown in the following example:

```
(defstruct-define-type type
          option-1
          option-2
          ...)
```

where each *option* is either the symbolic name of an option or a list of the form (*option-name* . *rest*). See the section "Options To **defstruct-define-type**", page 347.

Different options interpret *rest* in different ways. The symbol *type* is given an **si:defstruct-type-description** property of a structure that describes the type completely.

16.8.2 Options To defstruct-define-type

The documentation in this section regarding **defstruct** also applies to **zl:defstruct**.

:cons

The **:cons** option to **defstruct-define-type** is how you supply **defstruct** with the code necessary to cons up a form that constructs an instance of a structure of this type.

The **:cons** option has the syntax:

```
(:cons (inits description keywords) kind
       body)
```

body is some code that should construct and return a piece of code that constructs, initializes, and returns an instance of a structure of this type.

The symbol *inits* is bound to the information that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization:

- **:list** – *inits* is bound to a list of initializations, in the correct order, with **nils** in uninitialized slots.

- **:alist** – *inits* is bound to an alist with pairs of the form (*slot-number* . *init-code*).

The symbol *description* is bound to the instance of the **si:defstruct-description** structure that **defstruct** maintains for this particular structure. See the section "**defstruct** And **zl:defstruct** Internal Structures", page 351. This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* is bound to an alist with pairs of the form (*keyword* . *value*), where each *keyword* was a keyword supplied to the constructor macro that was not the name of a slot, and *value* was the Lisp object that followed the keyword. This is how you can make your own special keywords, such as the existing **:make-array** and **:times** keywords. See the section "Constructors For **defstruct** And **zl:defstruct** Structures", page 337. You specify the list of acceptable keywords with the **:keywords** option.

It is an error not to supply the **:cons** option to **defstruct-define-type**.

:ref

The **:ref** option to **defstruct-define-type** is how you supply **defstruct** with the necessary code that it needs to cons up a form that will reference an instance of a structure of this type.

The **:ref** option has the syntax:

```
(:ref (number description arg-1 ... arg-n)
      body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* is bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the **si:defstruct-slot-description** structure. See the section "**defstruct** And **zl:defstruct** Internal Structures", page 351.

The symbol *description* is bound to the instance of the **si:defstruct-description** structure that **defstruct** maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that is defaulted by the **:default-pointer** option. See the section "Options For **defstruct**

And **zl:defstruct**", page 321. **defstruct** checks that the user has supplied exactly *n* arguments to the accessor function before calling the reference consing code.

It is an error not to supply the **:ref** option to **defstruct-define-type**.

:overhead The **:overhead** option to **defstruct-define-type** is how you declare to **defstruct** that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of **:overhead** is: **(:overhead** *n***)** where *n* is a fixnum that says how many locations of overhead this type needs.

This number is used only by the **:size-macro** and **:size-symbol** options to **defstruct**. See the section "Options For **defstruct** And **zl:defstruct**", page 321.

:named The **:named** option to **defstruct-define-type** controls the use of the **:named** option to **defstruct**. With no argument, the **:named** option means that this type is an acceptable "named structure". With an argument, as in **(:named** *type-name***)**, the symbol *type-name* should be the name of some other structure type that **defstruct** should use if someone asks for the named version of this type. (For example, in the definition of the **:list** type the **:named** option is used like this: **(:named :named-list)**.)

:keywords The **:keywords** option to **defstruct-define-type** allows you to define additional constructor keywords for this type of structure. (The **:make-array** constructor keyword for structures of type **:array** is an example.) The syntax is: **(:keywords** *keyword-1* **...** *keyword-n***)**, where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist. See the section "Options To **defstruct-define-type**", page 347.

:defstruct The **:defstruct** option to **defstruct-define-type** allows you to run some code and return some forms as part of the expansion of the **defstruct** macro.

The **:defstruct** option has the syntax:

```
(:defstruct (description)
            body)
```

body is a piece of code that runs whenever **defstruct** is expanding a **defstruct** form that defines a structure of this type. The symbol *description* is bound to the instance of the **si:defstruct-description** structure that **defstruct** maintains for this particular structure.

The value returned by the *body* should be a *list* of forms to be included with those that the **defstruct** expands into. Thus, if you only want to run some code at **defstruct**-expand time, and you do not want to actually output any additional code, then you should be careful to return **nil** from the code in this option.

:predicate The **:predicate** option specifies how to construct a **:predicate** option for **defstruct**. The syntax for the option is:

```
(:predicate (description name)
            body)
```

The variable *description* is bound to the **si:defstruct-description** structure maintained for the structure for which a predicate is generated. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code that is evaluated to return the defining form for the predicate.

```
(:predicate (description name)
        `(defun ,name (x)
           (and (frobbozp x)
                (eq (frobbozref x 0)
                    ',(defstruct-description-name)))))
```

:copier The **:copier** option specifies how to copy a particular type of structure for situations when it is necessary to provide a copying function other than the one that **defstruct** would generate.

```
(:copier (description name)
  `(fset-carefully ',name 'copy-frobboz))
```

The syntax for the option follows.

```
(:copier (description name)
         body)
```

description is bound to an instance of the **si:defstruct-description** structure, *name* is bound to the symbol to be defined, and *body* is some code to evaluate to get the defining form.

16.9 defstruct And zl:defstruct Internal Structures

The documentation in this section regarding **defstruct** also applies to **zl:defstruct**.

If you want to write a program that examines structures and displays them the way **describe** and the Inspector do, your program will work by examining the internal structures used by **defstruct**. In addition to discussing these internal structures, this section also provides the information necessary to define your own structure types.

Whenever you use **defstruct** to define a new structure, **defstruct** creates an instance of the **si:defstruct-description** structure. This structure can be found as the **si:defstruct-description** property of the name of the structure; it contains such useful information as the name of the structure, the number of slots in the structure, and so on.

The following example shows a simplified version of how **si:defstruct-description** structure is actually defined. **si:defstruct-description** is defined in the **system-internals** (or **si**) package and includes additional slots that are not shown in this example:

```
;;;simplified version of si:defstruct-description structure
(cl:defstruct (defstruct-description
                (:default-pointer description)
                (:conc-name defstruct-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 **spaceship** or **phone-book-entry**.

The **size** slot contains the total number of locations in an instance of this kind of structure. This is *not* the same number as that obtained from the **:size-symbol** or **:size-macro** options 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 gets a number one larger than that stored in the **defstruct** description.

The **property-alist** slot contains an alist 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**. See the section "Options For **defstruct** And **zl:defstruct**", page 321.

The **slot-alist** slot contains an alist of pairs of the form (*slot-name . slot-description*). A *slot-description* is an instance of the **si:defstruct-slot-description** structure. The **si:defstruct-slot-description** structure is defined something like this, also in the **si** package:

```
;;;simplified version of the actual implementation
(cl:defstruct (defstruct-slot-description
                (:default-pointer slot-description)
                (:conc-name defstruct-slot-description-))
           number
           ppss
           init-code
           ref-macro-name)
```

Note that this is a simplified version of the real definition and does not fully represent the complete implementation. 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 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 the structure. See the section "Options To **defstruct-define-type**", page 347.

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 the symbol **si:%%defstruct-empty%%**.

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).

17. Flavors

The documentation on Flavors is organized in four areas:

We recommend that all users of Flavors read the sections noted as *Overview* and *Basic Concepts* above. Many Flavors-based programs do not require using any of the specialized programming practices described in *Advanced Concepts.*

If you do find that your program requires something extra, you can browse through the sections in *Advanced Concepts* to find the feature you are looking for.

The section in *Compatibility Features* describes message-passing, which is supported for compatibility with previous versions of Flavors. When writing new programs, it is good practice not to use message-passing. Because many system interfaces use message-passing, it is necessary to understand message-passing.

17.1 Basic Flavor Functions

This section describes several commonly used features, programming practices, and functions of Flavors.

17.1.1 Defining Flavors

The **defflavor** special form enables you to define flavors:

defflavor *name instance-variables component-flavors* &rest *options* *Special Form*

name is a symbol that is the name of this flavor. **defflavor** defines the name of the flavor as a type name in both the Common Lisp and Zetalisp type systems: See the section "Flavor Instances and Types", page 358.

instance-variables is a list of the names of the instance variables containing the local state of this flavor. Each element of this list can be written in two ways: either the name of the instance variable by itself, or a list containing the name of the instance variable and a default initial value for it. Any default initial values given here are forms that are evaluated by **make-instance** if they are not overridden by explicit arguments to **make-instance**.

If you do not supply an initial value for an instance variable as an argument to **make-instance**, and there is no default initial value provided in the **defflavor** form, the value of an instance variable remains unbound. (Another way to provide a default is by using the **:default-init-plist** option to **defflavor**.)

component-flavors is a list of names of the component flavors from which this flavor is built.

Each *option* can be either a keyword symbol or a list of a keyword symbol and its arguments. The options to **defflavor** are described elsewhere:

> See the section "Summary Of **defflavor** Options", page 355.
> See the section "Complete Options For **defflavor**", page 448.

Several of these options affect instance variables. These options can be given in two ways:

keyword	The keyword appearing by itself indicates that the option applies to all instance variables listed at the top of this **defflavor** form.
(*keyword var1 var2* ...)	A list containing the keyword and one or more instance variables indicates that this option refers only to the instance variables listed here.

The following form defines a flavor **wink** to represent tiddly-winks. The instance variables **x** and **y** store the location of the wink. The default initial value of both **x** and **y** is **0**. The instance variable **color** has no default initial value. The options specify that all instance variables are

:initable-instance-variables; **x** and **y** are **:writable-instance-variables**; and **color** is a **:readable-instance-variable.**

```
    (defflavor wink ((x 0) (y 0) color)      ;x and y represent location
              ()                             ;no component flavors
      :initable-instance-variables
      (:writable-instance-variables x y)     ;this implies readable
      (:readable-instance-variables color))
```

You can specify that an option should alter the behavior of instance variables inherited from a component flavor. To do so, include those instance variables explicitly in the list of instance variables at the top of the **defflavor** form. In the following example, the variables **x** and **y** are explicitly included in this **defflavor** form, even though they are inherited from the component flavor, **wink.** These variables are made initable in the **defflavor** form for **big-wink**; they are made writable in the **defflavor** form for **wink.**

```
    (defflavor big-wink (x y size)
              (wink)                         ;wink is a component
      (:initable-instance-variables x y))
```

If you specify a **defflavor** option for an instance variable that is not included in this **defflavor** form, an error is signalled. Flavors assumes you misspelled the name of the instance variable.

17.1.1.1 Summary Of defflavor Options

This section provides a brief summary of the options to **defflavor.** Each option is described fully elsewhere: See the section "Complete Options For **defflavor**", page 448.

The following options are used frequently:

:initable-instance-variables
Makes the specified instance variables *initable*, so you can initialize them when making an instance.

:readable-instance-variables
Makes the specified instance variables *readable*, by automatically creating an accessor function to read the value of each readable instance variable. If flavor *f* has a readable instance variable *v*, you can query an object for the value of *v* as follows:

(f-v object)

The accessor function is created in the current package.

:writable-instance-variables
Makes the specified instance variables *writable.* You can set the value of a writable instance variable by using **setf** and the accessor function (usually *f-v*):

```
(setf (f-v object) value)
```

All writable instance variables are also made readable.

:locatable-instance-variables
Enables you to get a locative pointer to the cell inside an instance that contains the value of an instance variable, using **locf** and the accessor function (usually *f-v*):

```
(locf (f-v object))
```

All locatable instance variables are also made readable.

:conc-name
Specifies a prefix for the accessor function created by the **:readable-instance-variables** option, which overrides the default (the name of the flavor followed by a hyphen).

:constructor
Generates a constructor function that creates new instances. These constructor functions are much faster than using **make-instance**. Constructor functions can accept positional arguments (instead of keyword arguments, like **make-instance**) or a mix of positional and keyword arguments. The constructor function is created in the current package.

:init-keywords
Declares its arguments as keywords that are accepted by **make-instance** of this flavor. Init keywords can be keyword arguments to be processed by methods defined for **make-instance** for this flavor, or keywords defined by the **:mixture** option to **defflavor**. See the section "Writing Methods For **make-instance**", page 365.

:default-init-plist
Provides a list of alternating initialization keywords and default value forms that are allowed for **make-instance** of this flavor. The keywords can be initable instance variables, init keywords, required init keywords, **:area**, or **:allow-other-keys**. You can specify any of these keywords as arguments to **make-instance** to override the default initial values given in the **:default-init-plist**.

:required-instance-variables
Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables.

:required-init-keywords
Specifies keywords that must be supplied (as an *init-option* to **make-instance** or in the **:default-init-plist** option to **defflavor**) when making an instance of this flavor.

:required-methods
Specifies generic functions that must be supported with methods by a flavor incorporating this one, if that flavor is intended to be instantiated.

:required-flavors
Specifies flavors that must be included as components (directly or indirectly) by any flavor incorporating this one, if that flavor is intended to be instantiated.

:method-combination
Declares the way that methods from different flavors are to be combined. See the section "Method Combination", page 420.

The following options are used less frequently:

:functions
Declares names of functions internal to the flavor. See the section "Defining Functions Internal to Flavors", page 443.

:mixture
Defines a family of related flavors. When **make-instance** is called, it uses its keyword arguments (or defaults to values in the **:default-init-plist** of the **defflavor** form) to choose which flavor of the family to instantiate.

:abstract-flavor
Declares that this flavor is not intended to be instantiated; it is often used for the base flavor of a flavor family.

:component-order
Explicitly states the ordering constraints on the flavor components whose order is important.

:documentation
Enables you to provides information on this flavor.

:area-keyword
Changes the keyword that specifies the area in which the instance is made; this keyword can then be given to **make-instance**. This is useful if the flavor is using the **:area** keyword for some other purpose, such as an init keyword for an object's geometric or geographic area.

:no-vanilla-flavor
Specifies that **flavor:vanilla** should not be included in this flavor.

:ordered-instance-variables
Specifies instance variables with fixed positions in the instance, for which speed is especially important. This increases efficiency at the cost of dynamic modification.

:method-order
Specifies names of generic functions for which speed is important. Flavors optimizes the speed for these functions.

The following options are available primarily for use by the system internals. When writing new programs, it is good practice not to use these options:

:special-instance-variables
Specifies instance variables that should be bound as special variables when a particular method is called. This option is used in conjunction with **:special-instance-variable-binding-methods**.

:special-instance-variable-binding-methods
Specifies certain methods that require the special instance variables to be bound as special variables. This option is used in conjunction with **:special-instance-variables**.

The following options are available for compatability with the previous flavor system. When writing new programs, it is good practice not to use these options:

:gettable-instance-variables
Enables automatic generation of methods for getting the values of instance variables, via messages. Use **:readable-instance-variables** instead.

:settable-instance-variables
Enables automatic generation of methods for setting the values of instance variables, via messages. Use **:writable-instance-variables** and **:initable-instance-variables** instead.

:default-handler
Specifies a function to be called when a generic function is called for which no method is available. This is the same as specifying a method for **:unclaimed-message**.

17.1.1.2 Flavor Instances and Types

defflavor defines the name of the flavor as a type name in both the Common Lisp and Zetalisp type systems. It is illegal for the name of a flavor to be the same as the name of an existing type (including a **defstruct** structure or a built-in type).

(**type-of** *instance*) and (**zl:typep** *instance*) return the symbol that is the name of the instance's flavor.

(**typep** *instance 'flavor-name*) and (**zl:typep** *instance 'flavor-name*) return **t** if the flavor of *instance* is named *flavor-name* or contains that flavor as a direct or indirect component, **nil** otherwise.

17.1.2 Defining Methods

defmethod *Special Form*

A method is the code that performs a generic function on an instance of a particular flavor. It is defined by a form such as:

(defmethod *(generic-function flavor options...) (arg1 arg2...)*
 body...)

The method defined by such a form performs the generic function named by *generic-function*, when that generic function is applied to an instance of the given *flavor*. (The name of the generic function should not be a keyword, unless you want to define a message to be used with the old **send** syntax.) You can include a documentation string and **declare** forms after the argument list and before the body.

A generic function is called as follows:

(*generic-function g-f-arg1 g-f-arg2...*)

Usually the flavor of *g-f-arg1* determines which method is called to perform the function. When the appropriate method is called, **self** is bound to the object itself (which was the first argument to the generic function). The arguments of the method are bound to any additional arguments given to the generic function. A method's argument list has the same syntax as in **defun**.

The *body* of a **defmethod** form behaves like the body of a **defun**, except that the lexical environment enables you to access instance variables by their names, and the instance by **self**.

For example, we can define a method for the generic function **list-position** that works on the flavor **wink**. **list-position** prints the representation of the object and returns a list of its x and y position.

```
(defmethod (list-position wink) () ; no args other than object
  "Returns a list of x and y position."
  (print self)                     ; self is bound to the instance
  (list x y))                      ; instance vars are accessible
```

The generic function **list-position** is now defined, with a method that implements it on instances of **wink**. We can use it as follows:

```
(list-position my-wink)
-->#<WINK 61311676>
   (4 0)
```

If no *options* are supplied, you are defining a primary method. Any *options* given are interpreted by the type of method combination declared with the **:method-combination** argument to either **defgeneric** or **defflavor**. See the section "Defining Special-Purpose Methods", page 430. For example, **:before** or **:after** can be supplied to indicate that this is a before-daemon or an after-daemon. For more information: See the section "Writing Before and After-Daemons", page 360.

If the generic function has not already been defined by **defgeneric**, **defmethod** sets up a generic function with no special options. If you call **defgeneric** for the name *generic-function* later, the generic function is updated to include any new options specified in the **defgeneric** form.

Several other sections of the documentation contain information related to **defmethod**:

See the section "**defmethod** Declarations", page 466.
See the section "Writing Methods For **make-instance**", page 365.
See the section "Function Specs for Flavor Functions", page 467.
See the section "Setter and Locator Function Specs", page 468.
See the section "Implicit Blocks for Methods", page 467.
See the section "Variant Syntax Of **defmethod**", page 467.
See the section "Defining Methods to Be Called by Message-Passing", page 471.

17.1.2.1 Writing Before and After-Daemons

This section describes how to use the default type of method combination, which is **:daemon :most-specific-first**. This section introduces concepts of method combination and ordering of flavor components that are covered in detail elsewhere:

See the section "Inheritance of Methods", page 377.
See the section "Ordering Flavor Components", page 382.

Daemon is the most commonly-used method combination in the system, and it is the easiest to use. Daemon method combination type lets you use *before-daemons* and *after-daemons*, which are two special types of methods. A before-daemon is a body of code that is to be executed before the primary method; an after-daemon is code to be executed after the primary method.

Typically, when you use **defmethod** to define a method to perform a generic function, you are defining a *primary* method. The primary method usually performs the main part of the generic function.

To write before-daemons and after-daemons, you write methods with the keywords

:before or **:after** as the *options* argument in the **defmethod** form. You need not specify the **:method-combination** option to **defflavor** or **defgeneric**, because **:daemon** is the default method combination type.

For example, a program with a **rocket** flavor might have a generic function called **launch**. The primary method for **launch** simply launches the rocket. You might define a before-daemon that puts in the fuel, and an after-daemon that activates the radar tracking, as follows:

```
(defflavor rocket ((tank :empty) (position :ground))
          ()
  :initable-instance-variables)

(defmethod (launch rocket) ()                    ;primary method
  (setq position :space))

(defmethod (launch rocket :before) ()            ;before-daemon
  ;;; add fuel before launching
  (setq tank :full))

(defmethod (launch rocket :after) ()             ;after-daemon
  ;;; activate radar tracking after launch
  (setq *radar-tracking* t))
```

When you use the **launch** generic function on an instance of **rocket** flavor, the flavor system uses a *combined method* to perform the operation. The combined method resembles:

```
(flavor:multiple-value-prog2 (before-daemon)
                             (primary-method)
                             (after-daemon))
```

flavor:multiple-value-prog2 ensures that the generic function returns any values returned by its primary method.

In this example, one flavor supplied a before-daemon, a primary method, and an after-daemon. **rocket** has no component flavors, so there were no other methods to consider.

When flavors are built from components, often one flavor provides a primary method, and others provide before or after-daemons to be combined with that primary method. In **:daemon** method combination, all available before-daemons are run, a single primary method is run, and all available after-daemons are run.

Suppose **flavl** is built on two component flavors:

```
(defflavor flav1 () (flav2 flav3))
(defflavor flav2 () ())       ;flav2 has no components
(defflavor flav3 () ())       ;flav3 has no components
```

If all three flavors provide a before-daemon, a primary method, and an after-daemon for the same generic function, the combined method for performing that generic function on **flavl** resembles:

```
(flavor:multiple-value-prog2
   (progn (before-daemon-flav1)
          (before-daemon-flav2)
          (before-daemon-flav3))
   (primary-method-flav1)
   (progn (after-daemon-flav3)
          (after-daemon-flav2)
          (after-daemon-flav1)))
```

The order in which the methods are executed in the combined method depends on the ordering of flavor components. Flavors computes the ordering of flavor components for **flavl** as:

```
(flav1 flav2 flav3)
```

flavl is said to be the *most-specific* flavor in the ordering, on the principle that specific flavors are usually built by including more general flavors as components. The **:daemon** method combination type states that before-daemons are executed in **:most-specific-first** method order, and after-daemons are executed in **:most-specific-last** method order. A single primary method is chosen from the set of available primary methods; it comes from the first flavor in the ordering that provides a primary method. For details on the ordering of flavor components: See the section "Ordering Flavor Components", page 382.

There can be only one before-daemon, one after-daemon, and one primary method written to implement a given generic function on any given flavor. If a second definition is given for an existing method, the previous method is overwritten and the new definition takes its place.

Flavors offers two advanced features that extend the flexibility of defining methods:

- Method Combination

  If the default way of method combination described here is not appropriate for your program, you can specify another type of method combination. See the section "Method Combination", page 420.

- Wrappers and Whoppers

When before and after-daemons are not powerful enough, you can use wrappers and whoppers, which let you put some code *around* the execution of the method: See the section "Wrappers and Whoppers", page 445.

17.1.2.2 Compiling Flavor Methods

When you are developing flavors-based programs, **compile-flavor-methods** is a useful tool. It causes the combined methods of a program to be compiled at compile-time, and the data structures to be built at load-time, rather than both happening at run-time. **compile-flavor-methods** is thus a very good thing to use, since the need to invoke the compiler at run-time slows down a program using flavors the first time it is run. You use **compile-flavor-methods** by including it in a file to be compiled.

See the macro **compile-flavor-methods** in *Symbolics Common Lisp: Language Dictionary.*

17.1.3 Making Instances of Flavors

make-instance *flavor-name* &rest *init-options* *Generic Function*

Creates and returns a new instance of the flavor named *flavor-name*, initialized according to *init-options*, which are alternating keywords and arguments. All *init-options* are passed to any methods defined for **make-instance.**

If **compile-flavor-methods** has not been done in advance, **make-instance** causes the combined methods of a program to be compiled, and the data structures to be generated. This is sometimes called *composing* the flavor. **make-instance** also checks that the requirements of the flavor are met. Requirements of the flavor are set up with these **defflavor** options: **:required-flavors**, **:required-methods**, **:required-init-keywords**, and **:required-instance-variables.**

init-options can include:

:initable-instance-variable value

You can supply keyword arguments to **make-instance** that have the same name as any instance variables specified as **:initable-instance-variables** in the **defflavor** form. Each keyword must be followed by its initial value. This overrides any defaults given in **defflavor** forms.

:init-keyword value

You can supply keyword arguments to **make-instance** that have the same name as any keywords specified as **:init-keywords** in the **defflavor** form. Each keyword must be followed by a value. This overrides any defaults given in **defflavor** forms.

:allow-other-keys t
Specifies that unrecognized keyword arguments are to be ignored.

:allow-other-keys :return
Specifies that a list of unrecognized keyword arguments are to be the second return value of **make-instance**. Otherwise only one value is returned, the new instance.

:area *number* Specifies the area number in which the new instance is to be created. Note that you can use the **:area-keyword** option to **defflavor** to change the **:area** keyword to **make-instance** to a keyword of your choice, such as **:area-for-instances**.

Note that any ancillary values constructed by **make-instance** (other than the instance itself) are constructed in whatever area you specify for them; this is not affected by using the **:area** keyword. For example, if you supply a variable initialization that causes consing, that allocation is done in whatever area you specify for it, not in this area. For example:

```
(defflavor foo ((foo-1 (make-array 100)))
           ())
```

In this example the array is consed in **sys:default-cons-area**.

:area nil Specifies that the new instance is to be created in the **sys:default-cons-area**. This is the default, unless the **:default-init-plist** option is used to specify a different default for **:area**.

If not supplied in the *init-options* argument to **make-instance**, the **:default-init-plist** option to the **defflavor** form is consulted for any default values for initable instance variables, init keywords, and the **:area** and **:allow-other-keys** options.

If you want to know what the allowed keyword arguments to **make-instance** are, use the Show Flavor Initializations command. See the section "Show Flavor Commands", page 399. `c-sh-A` works too, if the flavor name is constant.

You can define a method to run every time an instance of a certain flavor is created: See the section "Writing Methods For **make-instance**", page 365.

17.1.3.1 Writing Methods For make-instance

You can define a method to run every time an instance of a certain flavor is created, by defining a method for the generic function **make-instance**. This is a useful technique for performing additional initialization that depends on the instance variables being set to their initial values. If you write a method for **make-instance**, it is run after the instance is created and initialized according to *init-options*. Any values returned by **make-instance** methods are ignored.

Here is an example of a method defined for the **make-instance** function for the **freight-ship** flavor:

```
;;; Every time an instance of freight-ship is made,
;;; Load extra fuel to move the ship faster, if:
;;;    Cargo is perishable, and
;;;    We're in the hot season, and
;;;    The auxiliary fuel tank is empty.
(defmethod (make-instance freight-ship)
           (&key date-of-embarkation &allow-other-keys)
  (if (and (equal cargo-type :perishable)
           (hot-season-p date-of-embarkation)
           (equal auxiliary-fuel-tank :empty))
      (load-extra-fuel self)))
```

If a **make-instance** method allows an argument other than those that initialize instance variables, it is necessary to make that argument valid for **make-instance** of this flavor. To do so, use the **:init-keywords** option to **defflavor**. If the **make-instance** method requires any arguments, you should also use the **:required-init-keywords** option for **defflavor** to ensure that these arguments are always supplied. For example:

```
(defflavor freight-ship (cargo-type
                         primary-fuel-tank
                         auxiliary-fuel-tank)
           ()
  :initable-instance-variables
  (:init-keywords :date-of-embarkation)
  (:required-init-keywords :date-of-embarkation
                           :cargo-type
                           :auxiliary-fuel-tank))
```

make-instance methods receive all arguments that are given to **make-instance**. They also receive options not given to **make-instance** but present in the **:default-init-plist** option of the **defflavor** form (excluding options that initialize instance variables). Therefore, methods that use some of the arguments given to **make-instance** specify an argument list resembling:

(&key *arg1 arg2...* &allow-other-keys)

If you use &key, you must also use &allow-other-keys.

make-instance methods that do not use any of the arguments should specify **(&rest ignore)** as the argument list.

If a flavor has an **:init-keyword** named **foo** and an instance variable also named **foo**, you should use the more complex **&key** syntax:

```
(defmethod (make-instance f)
           (&key ((:foo foo-arg)) &allow-other-keys)
  body)
```

The generic function **make-instance** is implemented with the **:two-pass** method combination type. **:two-pass** means that the combined method executes all the primary methods and :after methods. **:after** methods are permitted because of unusual cases where a flavor's initializations must be performed after the initializations of its component flavors. **:before** methods are not allowed.

For any particular flavor, only one primary method and one **:after** method for **make-instance** are allowed. However, component flavors can supply primary methods and **:after** methods for **make-instance**; they are all run. When making an instance of a flavor that has component flavors, several of which have ordinary methods for **make-instance** and **:after** methods, the methods are executed as follows:

1. The instance is created and initialized according to *init-options*.

2. All primary methods for **make-instance** of this flavor and its component flavors are executed in most-specific-first order.

3. All **:after** methods for **make-instance** of this flavor and its component flavors are executed in most-specific-last order.

See the section "Built-in Types of Method Combination", page 423.

See the section "**:most-specific-first** And **:most-specific-last** Method Order", page 422.

For compatibility with previous versions of the flavor system, an **:init** message is also sent if there are any methods for it. Normally, this practice should be avoided in new programs; the exception to that guideline is for window flavors.

make-instance methods are executed before **:init** methods. If you use **:after :init** methods for window flavors, we recommend that you continue to use them instead of replacing them with **make-instance** methods. The window system is not yet capable of handling **make-instance** methods well because it still uses **:init** methods internally. If you use **make-instance** methods it would be possible that your methods would be executed before the system's **:init** methods, which could lead to problems.

17.1.4 Generic Functions

There are two kinds of functions in Symbolics Lisp: ordinary functions and generic functions.

Ordinary Functions	*Generic Functions*
Have a single definition.	Have a distributed definition.
Interface is specified by **defun**.	Interface can be specified by **defgeneric** or **defmethod**.
Implementation is specified by **defun**.	Implementation is specified by one or more **defmethod** forms.
Do not treat flavor instances specially.	First argument is usually an instance of a flavor.
Implementation is the same whenever the function is called.	Implementation varies from call to call, depending on the flavor of the first argument to the function.

Ordinary functions and generic functions are called with identical syntax:

(*function-name arguments...*)

The first argument to a generic function is an object. The flavor of the object determines which method is invoked to perform the generic function.

Generic functions are not only syntactically compatible with ordinary functions; they are semantically compatible as well:

- The name of a generic function is in a certain package and can be exported if it is part of an external interface. This allows programmers to keep unrelated programs separate.
- They are true functions that can be passed as arguments and used as the first argument to **funcall** and **mapcar**. For example:

```
(mapc #'reset counters)
```

- Program development tools such as **trace** can be used on generic functions.

Usually, the generic function chooses the appropriate method by looking at the flavor of its first argument. You can specify alternate means of dispatching by **defgeneric** and **define-method-combination**.

Generic functions replace message passing used in the old flavor system. However, message passing is still supported for compatibility with old programs. See the section "Using Message-Passing Instead of Generic Functions", page 471.

Do not remove the property **flavor:generic** from a generic function; this causes internal problems in the Flavors system.

17.1.4.1 Use Of defgeneric

It is not necessary to use **defgeneric** to write a generic function. If you define a method for an operation that has not been declared with **defgeneric, defmethod** automatically sets up a generic function with no special options.

defgeneric is used to:

- Formally define an interface.
- Specify options that pertain to the generic function as a whole.
- Gain the advantage of compile-time checking that the methods take the correct number of arguments, and that callers of the generic supply the correct number of arguments.
- Provide descriptive arguments to be displayed when you give the Arglist (m-X) command, or press c-sh-A for this generic function. (These default from the methods in a heuristic way if **defgeneric** is not used.)
- Document the contract of the generic function.

Here is an example of using **defgeneric** to document the contract of the generic function:

```
(defgeneric capacity (vehicle fare)            ;;; takes two args
  "Compute the number of passengers that can be
   accommodated at a given fare, and the number of
   crew required."
   (declare (values number-of-passengers
                    number-of-crew)))
```

You would use **capacity** as follows:

```
    (capacity my-ship 17.50)
```

In the unusual case when you want to define a generic function whose name is a keyword, you must use **defgeneric**. If you try to give a keyword as a generic function name using **defmethod**, it is interpreted as a message name, which must then be called with **send**. Note that defining a function (generic or ordinary) whose name is a keyword is not a recommended practice.

defgeneric *generic-function-name (arg1 arg2...) options...* *Special Form*

Defines a generic function named *generic-function-name* that accepts arguments defined by *(arg1 arg2...)*, a lambda-list. The first argument, *arg1*, is required, unless the **:function** option is used to indicate otherwise. *arg1* represents the object that is supplied as the first argument to the generic function. The flavor of *arg1* determines which method is appropriate to perform this generic function on the object. Any additional arguments (*arg2*, and so on) are passed to the methods.

The arguments to **defgeneric** are displayed when you give the Arglist (m-X) command or press c-sh-A while this generic function is current.

For example, to define a generic function **total-fuel-supply** that works on instances of **army** and **navy**, and takes one argument (*fuel-type*) in addition to the object itself, we might supply military-group as *arg1*:

```
(defgeneric total-fuel-supply (military-group fuel-type)
  "Returns today's total supply
    of the given type of fuel
    available to the given military group."
  (:method-combination :sum))
```

The generic function is called as follows:

```
(total-fuel-supply blue-army ':gas)
```

The argument **blue-army** is known to be of flavor **army**. Therefore, Flavors chooses the method that implements the **total-fuel-supply** generic function on instances of the **army** flavor. That method takes only one argument, *fuel-type*:

```
(defmethod (total-fuel-supply army) (fuel-type)
   body of method)
```

The set of *options* for **defgeneric** are described elsewhere: See the section "Options For **defgeneric**", page 370.

It is not necessary to use **defgeneric** to set up a generic function. For further discussion: See the section "Use Of **defgeneric**", page 368.

The function spec of a generic function is described elsewhere: See the section "Function Specs for Flavor Functions", page 467.

17.1.4.2 Options For defgeneric

Each *option* is either a keyword symbol, a list whose car is a keyword symbol and whose cdr is arguments to the option, a **declare** form, or a documentation string. The following options are accepted:

"*documentation*" A string specifying self-documentation, for the same purpose as a **defun** documentation string. This is the same as giving the **:documentation** option.

(**:compatible-message** *symbol*)
Enables you to invoke the methods by either calling the *generic-function-name* or sending the *symbol* message. *symbol* is the name of the message; it is usually a keyword, but need not be. This option is for compatibility with old Flavors and will eventually be made obsolete. For more information and an example of using this option: See the section "Defining a Compatible Message for a Generic Function", page 472.

(**declare** *declaration*)
Declarations that apply to the whole function (as opposed to declarations of variables) are permitted. These include **arglist**, **values**, **sys:downward-funarg**, **sys:function-parent**, and **optimize**, among others. You can repeat this option any number of times.

(**:dispatch** *name*) Specifies the name of an argument whose flavor controls selection of methods. The arguments seen by the methods are the arguments to the generic with *name* removed. Inside a method, **self** is bound to the object named by *name*. *name* is not mentioned explicitly in the **defmethod** arglist.

If **:dispatch** is not specified, the methods are selected on the basis of the flavor of the first argument to the generic function.

Dispatching off an argument other than the first argument is slightly slower because it involves an extra function call.

You can increase the efficiency by specifying (**:optimize speed**). This gains performance at the cost of flexibility in program development. If you do this, you can no longer change anything about the generic function without having to recompile callers. See the section on the **:optimize** option for **defgeneric**.

(**:documentation** "*documentation*")
A string specifying self-documentation. This is the same as giving a documentation string.

(:function *body...***)** Defines a function that runs instead of the generic dispatch. This is completely transparent to anyone calling the generic function. Such a prologue function can be used to rearrange the arguments, to standardize the arguments before the methods see them, to default optional arguments, to do the shared non-generic portion of an operation, or for any other purpose. To trigger the generic dispatch, apply the value of (**flavor:generic** *generic-function-name*) to the arguments. See the special form **flavor:generic** in *Symbolics Common Lisp: Language Dictionary*. Here is an example of its use:

```
(defgeneric size-of-value (type value)
  (:function
    (if (eq value *null-value*) 0
        (funcall (flavor:generic size-of-value)
                 type value))))
```

An important use of this option is to extend a generic function so it can also be used for objects that are not flavor instances.

:inline-methods Indicates that performance of methods for this generic function is crucial. This prevents the method combination from generating function calls from combined methods to the methods they combined. Instead, the methods are coded inline. The implementation of these methods trades space for time.

(:inline-methods :recursive)
Causes even more inline coding to occur. That is, if a method that is being coded inline contains a form **(f self args...)**, and **f** is a generic function with the **(:inline-methods :recursive)** option, then the entire **f** operation is also coded inline. It is not necessary that **f** be the same generic function as the one for which this method is being defined.

We do not recommend that **:inline-methods** be used automatically for all generic functions. One example of an appropriate use of **:inline-methods** is in the Table Management facility. The methods for each type of table are assembled from a large number of pieces contributed by different mixin flavors. The use of **:inline-methods** enhances the performance of this program.

Note that inline coding happens only for combined methods where Flavors knows exactly what the flavor of **self** is. In the following example, **test** could not be compiled inline in this method, since there is no way of knowing that the method **(flavor:method baz foo)** will not be inherited by a flavor that also inherits a different method for **test** than **(flavor:method test foo)**.

```
(defgeneric test (object)
  (:inline-methods :recursive))

(defflavor foo () ())

(defmethod (test foo) () body)

(defmethod (baz foo) ()
  (test self))
```

(**:method** (*flavor options...*) *body...*)

This option lets you define a method for this generic function in the **defgeneric** form, instead of in a separate **defmethod** form. It is simply a convenient abbreviation for (**defmethod** (*generic-function flavor options...*) *arglist body...*) The argument list for the method is computed from the argument list given at the top of the **defgeneric** form. This option can be repeated any number of times.

As a matter of style, **:method** is often used to define a default method; this is done by defining a method for the most basic member of a flavor family. You can include a documentation string or **declare** forms before *body*.

(**:method-arglist** *args...*)

You can use this option to specify that the methods accept different arguments than does the generic function itself. By default, the methods receive the same arguments that are specified at the top of the **defgeneric** form, except for the dispatching argument. The **:method-arglist** option is only meaningful in connection with the **:function** option.

In the following example, **:method-arglist** is used to declare that the *x* argument is optional for the generic function but required for the methods. The generic function **gf1** does not require the object itself to be passed as an argument.

```
(defvar *obj*)
(defflavor fl1 () ())
(setq *obj* (make-instance 'fl1))

(defgeneric gf1 (&optional obj x)
  (:function
    (unless obj (setq obj *obj*))
    (funcall (flavor:generic gf1) obj x))
  (:method-arglist x))

(defmethod (gf1 fl1) (x)
  (format t "GF1 FL1 called on ~S ~S" self x))
```

For a slightly more complex example, we can reverse the order of the arguments:

```
(defgeneric gf1 (&optional x obj)
  (:function
    (unless obj (setq obj *obj*))
    (funcall (flavor:generic gf1) obj x))
  (:method-arglist x))
```

Doing **(gf1)** does the same thing as before, but now you can also do **(gf1 23)**.

(:method-combination *name args...*)

Specifies the type of method combination to be used in handling this generic function. If this option is used, all flavors must use the same method combination for this generic function. If the **:method-combination** option is also supplied to **defflavor**, that option must agree with the **:method-combination** option given to **defgeneric**. The default is **(:method-combination :daemon :most-specific-first)**. For more information on usage and an example: See the section "Using The **:method-combination** Option", page 420.

(:optimize speed) Increases the speed of the generic function when dispatching off an argument other than the first argument. This has an effect only if **:dispatch** is also specified. The package of the symbol **speed** is unimportant; this syntax is consistent with the Common Lisp syntax for the **optimize** declaration.

If you use **(:optimize speed)** in conjunction with **:dispatch**, callers of the generic function are responsible for including the code to rearrange the arguments. The compiler puts in this code. Note that you must recompile all callers if you change anything about the generic function.

17.1.5 Redefining Flavors, Methods, and Generic Functions

You can redefine flavors, methods, and generic functions at any time. To do so, simply evaluate another **defflavor, defmethod**, or **defgeneric** form. The new definition replaces the old. This flexibility is useful in program development.

17.1.5.1 Redefining a Flavor

You can redefine a flavor by editing its **defflavor** form and then compiling the new definition, either by using the Zmacs command c-sh-C, or the **recompile-flavor** function.

Sometimes redefining a flavor causes old instances to be outdated; for example, adding or removing instance variables, or changing the order of instance variables. In these cases, Flavors gives you a warning that the flavor was changed in such a way that the stored representation is different. However, this does not cause a problem. When old instances are next accessed, they are updated to the new format. New instance variables will be initialized if the **defflavor** form indicates a default value, or left unbound otherwise. When a flavor is changed, the Flavors system propagates the changes to any flavors of which it is a direct or indirect component.

You can use **flavor:rename-instance-variable** to give an instance variable a new name, and to ensure that its value is preserved, for existing instances.

You can remove the definition of a flavor by using the Zmacs command Kill Definition (m-X), or the **flavor:remove-flavor** function.

17.1.5.2 Changing an Instance

You can explicitly change an existing instance in these ways: evaluate a new **defflavor** form; rename one or more of its instance variables using **flavor:rename-instance-variable**; or change the flavor of the instance using **change-instance-flavor**. If you redefine a flavor that has already been instantiated, this implicitly causes existing instances to be updated; this is described above. When you change instances, you should consider possible side effects; for example, any methods written for **make-instance** do not run when you change an instance. If you need to perform further initialization when an instance is changed, use **flavor:transform-instance**. See the generic function **flavor:transform-instance** in *Symbolics Common Lisp: Language Dictionary.*

17.1.5.3 Redefining Generic Functions

Usually, you can redefine a generic function to be an ordinary function, or an ordinary function to be a generic function, without having to recompile any callers. However, if you use **defgeneric** and specify the **:dispatch** and

(:optimize speed) options, you must recompile callers if you redefine the generic function.

Do not use **fundefine** to remove the definition of a generic function. If you do so, and then compile a **defmethod** form, the generic function remains undefined until you do an explicit **defgeneric**. While the generic function is undefined, any callers to it will malfunction. Also, do not remove the property **flavor:generic** from a generic function; this causes internal problems to Flavors.

17.1.5.4 Redefining Wrappers and Whoppers

Whoppers are functions, not macros, so they can be redefined at any time; the new definition replaces the old.

Redefining a wrapper automatically performs the necessary recompilation of the combined method of the flavor. If the wrapper is given a new definition, the combined method is recompiled so that it gets the new definition. If a wrapper is redefined with the same old definition, the existing combined methods continue to be used, since they are still correct. The old and new definitions are compared using the function **equal**.

Because **defwhopper-subst** defines a wrapper, issues with redefining them are the same as for wrappers.

Related Functions:

change-instance-flavor	Changes the flavor of an instance.
recompile-flavor	Updates the internal data of the flavor and any flavors that depend on it.
flavor:remove-flavor	Removes the definition of a flavor.
flavor:rename-instance-variable	Changes the name of an instance variable, carrying the value of the old instance variable to the new for any existing instances.
flavor:transform-instance	Executes code when an instance is changed to a new flavor; thus enables you to perform initialization of the instance. Use this generic function by defining methods for it. It is not intended to be called.

Related Editor Tools:

Zmacs offers a variety of tools for redefining flavors, generic functions, and methods: See the section "Zmacs Commands for Flavors, Generic Functions, and Methods", page 410.

17.2 Mixing Flavors

It is advantageous to mix flavors when the characteristics of two or more different kinds of objects overlap. You identify the common characteristics and define a flavor that can be incorporated (or mixed) into both kinds of objects. This is called a *component flavor.*

For example, we might write a program involving space-ships and comets. The characteristics of space-ships and comets partially overlap; they both have x, y, and z-velocity. We can define a flavor called **3-d-moving-object** that represents their common characteristics:

```
(defflavor 3-d-moving-object (x-velocity y-velocity z-velocity)
           ()
  :initable-instance-variables)
```

Once **3-d-moving-object** is defined, we can include it in the definition of the flavors **space-ship** and **comet** as follows:

```
(defflavor space-ship (crew-list name destination)
           (3-d-moving-object)        ;component flavor
  :initable-instance-variables)

(defflavor comet (percent-iron estimated-mass)
           (3-d-moving-object)        ;component flavor
  :initable-instance-variables)
```

Objects of the **space-ship** and **comet** flavors inherit the instance variables of their component flavor **3-d-moving-object.** When making an instance of **comet** we can initialize its inherited variables:

```
(make-instance 'comet :estimated-mass 27
                      :x-velocity 12
                      :y-velocity 44
                      :z-velocity 87)
```

When you use existing flavors to create a new flavor, the new flavor inherits characteristics of each of its component flavors (and of its components' components, and so on). This includes instance variables, methods, and other characteristics that are attached to a **defflavor** form, such as **:default-init-plist.**

Typically, a program has a family of related flavors. The most basic flavor has instance variables and methods defined that relate to the whole family of flavors. Flavors built on the basic flavor inherit those instance variables and methods, and include additional (more specialized) instance variables and methods. See the section "Flavor Families", page 386.

17.2.1 Inheritance of Methods

When a generic function is applied to an object of a particular flavor, methods for that generic function attached to that flavor or to its components are available. From this set of available methods, one or more are selected to be called. If more than one is selected, they must be called in some particular order and the values they return must be combined somehow.

Flavors constructs a single *handler* for each generic function supported by the new flavor. The handler is the code that actually performs the generic function on an instance of the flavor. In the simplest case, the handler is simply one of the methods.

17.2.1.1 Example of a Combined Method

In other cases the handler is a *combined method.* One example of a combined method is the handler constructed when the default method combination type (**:daemon**) is used, and a single **:before** method and **:after** method is present. The body of the combined method resembles:

```
(flavor:multiple-value-prog2 (before-method)
                             (primary-method)
                             (after-method))
```

17.2.1.2 Definition of Method Combination

The way that Flavors constructs a handler is called *method combination.* Often many methods are defined for performing a generic function on objects of a given flavor (some of the methods are defined for component flavors), and these methods must somehow be combined. The way that methods are combined depends on two factors:

- The designated type of method combination.

  Each generic function is associated with a type of method combination. By default this is **:daemon.** You can use the **:method-combination** option to **defflavor** or **defgeneric** to specify a different mechanism.

- The designated method order.

  By default this is **:most-specific-first.** In most types of method combinations you can specify **:most-specific-last** to the **:method-combination** option.

The type of method combination and the method order are sufficient to choose which methods from the set of available methods should be run, and in what order they are to be run.

For example, the **:and** method combination type chooses all available primary methods and combines them inside an **:and** special form. Any **:before** or **:after** methods cause an error.

The order in which the methods are executed is important. This is determined by the ordering of flavor components and the method order used. When flavors are built from components, the flavor system computes a total ordering of components. For example, three flavors are defined as follows:

```
(defflavor flav-1 () (flav-2) (flav-3))
(defflavor flav-2 () ())
(defflavor flav-3 () ())
```

The ordering of flavor components for the flavor **flav-1** is:

```
(flav-1 flav-2 flav-3)
```

flav-1 is said to be the most specific flavor in the ordering. Thus if **:most-specific-first** method order is used with the **:and** method combination type, and all three flavors define a method for the same generic function, the combined method resembles:

```
(and (method-for-flav-1)
     (method-for-flav-2)
     (method-for-flav-3))
```

If **:most-specific-last** method order is used with the **:and** method combination type, the combined method resembles:

```
(and (method-for-flav-3)
     (method-for-flav-2)
     (method-for-flav-1))
```

For information on how the flavor ordering is computed: See the section "Ordering Flavor Components", page 382.

For information on other built-in types of method combination, and how to define new types of method combination: See the section "Method Combination", page 420.

17.2.2 Inheritance of Instance Variables

When you define a flavor that is built on other flavors, the new flavor inherits instance variables from each of its component flavors. When two or more of the components have an instance variable with the same name, the new flavor inherits exactly one instance variable with that name. All components of the new flavor share this variable. The default initial value for an instance variable comes from the first flavor in the ordering of flavor components that specifies a value. For example:

```
;;; the basic flavor in the family of printers
;;; all printers must have a name and baud-rate
(defflavor basic-printer (name baud-rate)
           ()
  :initable-instance-variables
  (:required-init-keywords name baud-rate))

;;; line-printers in this shop run at 1200 baud
(defflavor line-printer ()
           (basic-printer)          ;built on basic-printer
  (:default-init-plist :baud-rate 1200))

;;; smart printers know their own status and length of queue
;;; in this shop they run at 2400 baud
(defflavor smart-printer (current-status queue)
           (basic-printer)          ;built on basic-printer
  :initable-instance-variables
  (:default-init-plist :baud-rate 2400
                       :queue 0))
```

In this flavor family, all types of printers have instance variables for **name** and **baud-rate**. The flavors **smart-printer** and **line-printer** inherit the instance variables **name** and **baud-rate** from their component flavor **basic-printer**.

The flavor **basic-printer** specifies that the **name** and **baud-rate** instance variables are required to be initialized, when making an instance of **basic-printer** or either of the two flavors that are built on it.

The flavor **line-printer** has only one difference from **basic-printer**; it specifies a default initial value for the **baud-rate** instance variable.

The flavor **smart-printer** supplies two additional instance variables, **current-status** and **queue**. It also specifies a default initial value for **queue** and for **baud-rate**.

17.2.3 Inheritance Of defflavor Options

A flavor built from components inherits characteristics from those components, including many of the **defflavor** options. This section describes which **defflavor** options are inherited, and which are not.

:abstract-flavor	Not inherited.
:area-keyword	Inherited. If more than one component specifies this option, the **:area-keyword** chosen comes from the most specific flavor in the ordering.

:component-order	Inherited. Any ordering restrictions noted in a flavor's **:component-order** option are considered as ordering restrictions by all flavors that are built on it.
:conc-name	Not inherited.
:constructor	Not inherited.
:default-handler	Inherited.
:default-init-plist	Inherited. A flavor's **:default-init-plist** is the union of its own, and those of its components. If any elements of the **:default-init-plist** are in conflict, the conflict is resolved by choosing the element from the most specific flavor that provides it.
:documentation	Not inherited.
:functions	Inherited.
:gettable-instance-variables	Inherited. Any instance variables specified as gettable by a flavor component are also gettable by the flavor that is built on that component.
:init-keywords	Inherited. A flavor's list of **:init-keywords** is the union of the **:init-keywords** of all of its components.
:initable-instance-variables	Inherited. Any instance variables specified as initable by a flavor component are also initable by the flavor that is built on that component.
:locatable-instance-variables	Inherited. Any instance variables specified as locatable by a flavor component are also locatable by the flavor that is built on that component.
:method-combination	Inherited. If **:method-combination** is specified more than once for the same generic function by flavor components, they must agree on the type of method combination and the parameters.
:method-order	Inherited. A flavor's method order is the resulting of appending the method-orders of all its components, in the order of flavor components.
:mixture	Not inherited.
:no-vanilla-flavor	Inherited.

:ordered-instance-variables
Inherited. If this is specified by more than one component, the order of the instance variables must agree. For example, if one flavor specifies (**:ordered-instance-variables a b c**), another component flavor could legally specify (**:ordered-instance-variables a b c d e**). However it would be illegal for a component flavor to specify (**:ordered-instance-variables d e**).

:readable-instance-variables
Inherited. Any instance variables specified as readable by a flavor component are also readable by the flavor that is built on that component.

:required-flavors Inherited. A flavor's list of **:required-flavors** is the union of the **:required-flavors** of all of its components.

:required-init-keywords
Inherited. A flavor's list of **:required-init-keywords** is the union of the **:required-init-keywords** of all of its components.

:required-instance-variables
Inherited. A flavor's list of **:required-instance-variables** is the union of the **:required-instance-variables** of all of its components.

:required-methods
Inherited. A flavor's list of **:required-methods** is the union of the **:required-methods** of all of its components.

:settable-instance-variables
Inherited. Any instance variables specified as settable by a flavor component are also settable by the flavor that is built on that component.

:special-instance-variables
Inherited. A flavor's list of **:special-instance-variables** is the union of the **:special-instance-variables** of all of its components.

:special-instance-variable-binding-methods
Inherited. A flavor's list of **:special-instance-variable-binding-methods** is the union of the **:special-instance-variable-binding-methods** of all of its components.

:writable-instance-variables
Inherited. Any instance variables specified as writable by a flavor component are also writable by the flavor that is built on that component.

17.2.4 The Vanilla Flavor

By default, every flavor includes the flavor called **flavor:vanilla**. **flavor:vanilla** has no instance variables, but it provides several basic useful methods, some of which are used by the Flavor tools. See the section "Generic Functions and Messages Supported By **flavor:vanilla**", page 470.

You can specify not to include **flavor:vanilla** by providing the **:no-vanilla-flavor** option to **defflavor**. This would be unusual.

17.2.5 Ordering Flavor Components

This section describes how Flavors determines the ordering of flavor components, when a new flavor is built on component flavors. You can view the order using Show Flavor Components: See the section "Show Flavor Commands", page 399.

flavor:get-all-flavor-components returns the list of ordered components: See the function **flavor:get-all-flavor-components** in *Symbolics Common Lisp: Language Dictionary*.

The **defflavor** forms of a flavor and its components set local constraints on the ordering of flavor components. When a flavor is built from components, all of the local constraints of the flavor and its components are taken into account, and an ordering is computed that satisfies all of these constraints. In other words, the **defflavor** forms specify partial orderings, which must be merged into one total ordering.

Three rules govern the ordering of flavor components:

1. A flavor always precedes its own components.

2. The local ordering of flavor components is preserved.

3. Duplicate flavors are eliminated from the ordering; if a flavor appears more than once, it is placed as close to the beginning of the ordering as possible, while still obeying the other rules.

In many cases, these three rules are enough to define a unique ordering of flavor components. In other cases, the rules result in several possible orderings, and Flavors applies another guideline:

A tree of flavors is constructed from the list of flavor components. The ordering is determined by walking through the tree of flavors in depth-first order, and adding each node (flavor) to the ordering if it fits the constraints of the three rules.

The first node (the root of the tree) is the first flavor in the ordering, provided that it does not transgress any of the rules. The Flavor system continues to the next node, traversing the tree in depth-first order. If the node can be placed next in the ordering without transgressing one of the three rules, it is added it to the ordering. If adding it would transgress a rule, that node is skipped. If the end of the tree is reached and some flavors have not yet been placed in the ordered list, the Flavor system walks through the tree again, applies the three rules to the remaining nodes and adds them to the ordering. In complicated programs, it is conceivable that we could walk through the tree several times.

The following examples illustrate how we could predict how new Flavors would choose an ordering of components of **pie**, which is defined as follows:

17.2.5.1 Simple Example of Ordering Flavor Components

In this example, the three rules define exactly one ordering for **new-flavor**:

```
(defflavor new-flavor () (apple tomato))
(defflavor apple () (fruit))
(defflavor tomato () (fruit vegetable))
(defflavor fruit () ())
(defflavor vegetable () ())
```

To illustrate how the rules apply to this example:

1. A flavor always precedes its own components.

    - **new-flavor** precedes **apple** and **tomato.**
    - **apple** precedes **fruit.**
    - **tomato** precedes **fruit** and **vegetable.**

2. The local ordering of flavor components is preserved.

    - **apple** precedes **tomato.**
    - **fruit** precedes **vegetable.**

The **flavor:vanilla** flavor is included in all flavors unless it is explicitly excluded (by the **:no-vanilla-flavor** option to **defflavor**). The first rule constrains **flavor:vanilla** to appear last in every ordering of flavors, if it is present.

The resulting ordering of flavors is:

```
(new-flavor apple tomato fruit vegetable flavor:vanilla)
```

17.2.5.2 Using The Tree to Order Components

In this example, the three rules do not define a single ordering for **pie**. The flavors are defined as follows:

```
(defflavor pie () (apple cinnamon))
(defflavor apple () (fruit))
(defflavor cinnamon () (spice))
(defflavor fruit () ())
(defflavor spice () ())
```

Three orderings are possible, under the rules:

```
(pie apple fruit cinnamon spice flavor:vanilla)
(pie apple cinnamon fruit spice flavor:vanilla)
(pie apple cinnamon spice fruit flavor:vanilla)
```

A well-conceived program should not depend on any one of those orderings, but should work equally well under any of them. For example, if your program depends on **spice** preceding **fruit** in the ordering for **pie**, you should make that constraint explicit, by including those two flavors in the list of components in the **defflavor** form for **pie**.

The following text describes how to predict the ordering that the Flavors system would choose, when several orderings are possible. We construct a tree of flavor components:

```
          pie
         /   \
      apple  cinnamon
       /          \
     fruit      spice
```

We order the components by walking through the tree in a depth-first order, adding the components to the order, always checking that no rules are transgressed. The resulting ordering is:

```
(pie apple fruit cinnamon spice flavor:vanilla)
```

17.2.5.3 Example of an Inconsistent Set of Flavor Definitions

It is possible to write a set of flavor definitions that cannot be ordered using the rules. For example:

```
(defflavor new-flavor () (fruit apple))
(defflavor apple () (fruit))
```

To illustrate how the rules apply to this example:

1. A flavor always precedes its own components.

    - **new-flavor** precedes **fruit** and **apple.**
    - **apple** precedes **fruit.**

2. The local ordering of flavor components is preserved.

    - **fruit** precedes **apple**

Two of the rules contradict each other: **apple** precedes **fruit**, and **fruit** precedes **apple**. When this situation occurs, Flavors signals an error when it tries to compute the ordering (usually the first time you call **compile-flavor-methods** or **make-instance**). The error message includes a detailed description of the minimal set of conflicting constraints that cause the inconsistency. At that point you can redefine one or more of the flavors to resolve the problem, either by changing the order of flavor components so there is no conflict, or by using the **:component-order** option to **defflavor** to relax the ordering constraints.

17.2.5.4 Skipping a Flavor When Traversing the Tree

This example illustrates the case when a flavor cannot be added to the ordering the first time it appears in the tree:

```
(defflavor pie () (apple cinnamon))
(defflavor apple () (fruit))
(defflavor cinnamon () (spice))
(defflavor fruit () (food))
(defflavor spice () (food))
(defflavor food () ())
```

We construct a tree of flavors:

```
          pie
         /   \
     apple  cinnamon
     /          \
   fruit       spice
   /              \
 food             food
```

The list begins with **pie**. We continue, adding **apple** and **fruit**. So far, the (incomplete) ordered list is:

```
(pie apple fruit
```

The next node is **food**, but we cannot place it next in the ordering because doing so would transgress the rule that a flavor always precedes its own components. If placed next, **food** would precede **spice**, and we know that **spice** must precede **food**. Thus we skip **food** and continue through the tree. Because **food** appears later in the tree, we pick it up then. The ordering is now complete:

```
(pie apple fruit cinnamon spice food flavor:vanilla)
```

In more complex cases, we might finish walking through the tree without picking up all the flavors. In that case, we would walk through the tree again, picking up the remaining flavors as long as doing so would not transgress the rules. It is conceivable that we might need to walk through the tree several times, if many related flavors are mixed.

17.2.5.5 Final Note

Note the following example:

```
(defflavor pie () (apple cinnamon))
(defflavor pastry () (cinnamon apple))
```

The ordering for **pie** is:

```
(pie apple cinnamon flavor:vanilla)
```

The ordering for **pastry** is:

```
(pie cinnamon apple flavor:vanilla)
```

There is no problem with the fact that **apple** precedes **cinnamon** in the ordering of the components of **pie**, but not in the ordering for **pastry**. However, you cannot build a new flavor that has both **pie** and **pastry** as components.

17.2.6 Flavor Families

The following organization conventions are recommended for programs that use flavors:

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have that base flavor as one of their components. Typically the base flavor includes things relevant to the whole family, such as instance variables, **:required-methods** and **:required-instance-variables** declarations, default methods for certain generic functions, **:method-combination** declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most cannot be instantiated and merely serve as a foundation on 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. You construct usable flavors by choosing the mixins for the desired characteristics and combining them with the base flavor. A mixin flavor that provides the *mumble* feature is often named *mumble*-**mixin**.

An instantiatable flavor combines several mixins with a base flavor to produce desired behavior. It is a complete description that can be used with **make-instance**.

By organizing your flavors 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 components controls the order in which methods are combined. Order dependencies should be documented as part of the conventions of the appropriate family of flavors.

You can follow these conventions by naming the basic flavors and mixin flavors as suggested, and using **defflavor** to define usable flavors composed of a handful of mixins and the basic flavor. **defflavor** also offers several options that can help organize flavor families:

- **:abstract-flavor** specifies that a flavor cannot be instantiated.
- **:component-order** states explicitly the order restrictions you want imposed on component flavors. When many mixins are combined together, the normal ordering constraints can cause conflicts that make it impossible to compute a total ordering of flavor components. If you know that several of the components should have ordering restraints but that others need not, you can relax the ordering restrictions by using **:component-order**.
- **:documentation** lets you document the organization of the flavor family.
- **:mixture** lets you write a framework with rules that define a flavor family. This is a more structured way to organize a flavor family.
- **:required-methods**, **:required-instance-variables**, and **:required-flavors** let you explicitly state the requirements of flavors that are to be instantiated.

17.3 Example of Programming with Flavors: Life

This section contains an annotated program that illustrates several principles of using Flavors. (This code is in `sys:examples;flavor-life.lisp`.) Because Flavors is intended to help organize large, complex programs, it is difficult to exercise all (or even most) of the features of Flavors in a small demonstration program. However, this program does illustrate the following aspects of Flavors:

- Using **defflavor**, **defgeneric**, **defmethod**, and **make-instance**
- Writing a method for **make-instance**
- Constructing a flavor from component flavors
- Using **:and** type of method combination
- Integrating flavors with other Lisp data structures (arrays)
- Using message-passing (**send**) when necessary to use old interfaces

The program is called Life; it is a game that simulates a community of beings called cells. The rules of Life indicate whether a given cell will live or die in the next generation, depending on its environment. If the cell is too crowded by its neighbors, or too isolated from other cells, it dies. Otherwise the environment is deemed acceptable, and the cell lives. Specifically:

- Cells live in a two-dimensional array.
- A cell typically has eight neighbors, those adjacent to it.
- Cells on the border of the array have less than eight neighbors.
- If an empty cell has exactly 3 live neighbors, a cell is born there.
- If an empty cell has any other number of live neighbors, no cell is born.
- If a live cell has 2 or 3 live neighbors, it stays alive.
- If a live cell has any other number of live neighbors, it dies.

Note that this example program is intended to illustrate Flavors, and is not attempting to run Life in the fastest possible way.

17.3.0.1 Organizing the Program

Three flavors are used to represent the Life cells: **cell**, **box**, and **box-with-cell**. The **cell** flavor is flexible enough that you could use it in another program that uses a different type of display. The **box** flavor is used to display the Life gameboard on the screen. **box-with-cell** combines those two flavors.

The following illustration shows the Life gameboard implemented by this program. In this illustration, the variable ***number-boxes-on-axis*** is 3; that is, the Life game is played on a 3 by 3 array. (In the program below, ***number-boxes-on-axis*** is much larger than 3.) Each instance of **box-with-cell** is a unit of the Life game, and displayed in each generation. To make it easier to compute live neighbors of each **box-with-cell**, we have an invisible border composed of instances of **cell** (all dead). This way, each **box-with-cell** has eight neighbors. The border cells are not displayed. Only the inner part of the gameboard is displayed for the user.

Life Gameboard

```
+--------+-------+-------+-------+-------+
|  dead  | dead  | dead  | dead  | dead  |
|  cell  | cell  | cell  | cell  | cell  |
+--------+-------+-------+-------+-------+
| dead   | box-  | box-  | box-  | dead  |
| cell   | with- | with- | with- | cell  |
|        | cell  | cell  | cell  |       |
+--------+-------+-------+-------+-------+
| dead   | box-  | box-  | box-  | dead  |
| cell    | with- | with- | with- | cell  |
|         | cell  | cell  | cell  |       |
+---------+-------+-------+-------+-------+
| dead    | box-  | box-  | box-  | dead  |
| cell    | with- | with- | with- | cell  |
|         | cell  | cell  | cell  |       |
+---------+-------+-------+-------+-------+
|  dead   | dead  | dead  | dead  | dead  |
|  cell   | cell  | cell  | cell  | cell  |
+---------+-------+-------+-------+-------+
```

17.3.0.2 Defining Variables Used in the Program

The program begins by defining and initializing special variables:

```
;;; This array is used for storing cells and box-with-cells.
(defvar *game-board*  nil
  "array for the Life game.")
(defvar *number-boxes-on-axis*  30
  "number of boxes on each axis of gameboard")

;;; This window is used for displaying the *game-board*.
(defvar *game-window*  (tv:make-window 'tv:window
                                       :blinker-p nil
                                       :label "Game of Life")
  "Window for display.")

;;; The following numbers make a nice display:
(defvar *x-top-left-corner* 250
   "x coordinate of top left corner of display")
(defvar *y-top-left-corner* 100
  "y coordinate of top left corner of display")
(defvar *side-length* 15
  "length of each box for display")
(defvar *board-length*  (* *side-length* *number-boxes-on-axis*)
  "length of gameboard for display")
(defvar *cell-radius* 6
  "radius of circle to draw live cells")
```

17.3.0.3 Defining the Flavors

```
;;; A cell is a functional cell of the Life game.  It stores
;;; its status and next-status (:alive or :dead), and a list of
;;; its neighbors.   The x and y instance variables give the
;;; coordinates of this cell in the array *game-board*.

(defflavor cell (x y status next-status neighbors) ()
  (:initable-instance-variables x y status))

;;; A box is intended only to be displayed on the screen.
;;; box-x and box-y are the coordinates of its top-left corner.
;;; side-length is the length of one side of the box.

(defflavor box (box-x box-y box-x-center box-y-center side-length)
           ()
  (:initable-instance-variables box-x box-y side-length))

;;; A box-with-cell is a box that contains a functional Life cell.
;;; The cell component provides the necessary methods for playing Life.
;;; The box component makes it easy to display the game-board.
;;; box-with-cell has default initial values for our game.

(defflavor box-with-cell ()                    ;no instance vars
           (box cell)                          ;two components
  (:default-init-plist :side-length *side-length*
                       :status (if (evenp (random 2)) ':alive ':dead))
  (:required-methods aliveness count-live-neighbors
                     get-next-status change-status
                     draw-outline))
```

17.3.0.4 Defining th eneric Functions

Now we define the generic functions. The argument *cell-unit* implies that the generic function can be used on instances of **cell** or any flavor built on **cell**, such as **box-with-cell**. Similarly, the argument *box-unit* implies that the generic function **draw-outline** can be used on instances of **box** or **box-with-cell**. We use the **defgeneric** forms as the appropriate place to document the contract of the generic function. (One additional generic function, **change-status**, is defined later on in this section.)

```
(defgeneric aliveness (cell-unit)
  "Returns 1 if the cell-unit is currently alive, 0 otherwise.")

(defgeneric find-neighbors (cell-unit)
  "Calculates and stores the 8 neighbors of a cell-unit.")

(defgeneric count-live-neighbors (cell-unit)
  "Returns the number of live neighbors of a cell-unit.")
```

```
(defgeneric get-next-status (cell-unit)
  "Applies rules of the Life game, using count-live-neighbors.
   If overcrowded or too isolated, this cell-unit dies.
   If the environment is fine, this cell-unit lives.
   The result is remembered by the cell-unit."

(defgeneric draw-outline (box-unit)
  "Draws the outline of the given box-unit.")

(defgeneric draw-contents (box-with-cell)
  "Draws the cell contained in a box-with-cell.
   Live cells appear as filled-in circles; dead cells are invisible.")
```

17.3.0.5 Defining the Methods

We now define the methods that implement the generic functions on instances of **cell** and **box**. These methods are inherited by **box-with-cell**. Notice that the methods for **draw-outline** and **draw-contents** use message-passing to invoke methods associated with the window system. There is no difficulty in writing programs that use both generic functions and message-passing. (The methods for **change-status** are defined later on in this section).

```
;;; aliveness returns 1 if cell is alive, 0 otherwise

(defmethod (aliveness cell) ()
  (if (eq status ':alive) 1 0))

;;; the neighbors of a cell are its 8 adjacent cells
;;; because there is a border of dead cells, in this implementation
;;; every box-with-cell has 8 neighbors.

(defmethod (find-neighbors cell) ()
  (setq neighbors (list
                    (aref *game-board* x (1- y))
                    (aref *game-board* x (1+ y))
                    (aref *game-board* (1- x) y)
                    (aref *game-board* (1+ x) y)
                    (aref *game-board* (1- x) (1- y))
                    (aref *game-board* (1- x) (1+ y))
                    (aref *game-board* (1+ x) (1- y))
                    (aref *game-board* (1+ x) (1+ y)))))

(defmethod (count-live-neighbors cell) ()
  (reduce #'+ (map 'list #'aliveness neighbors)))

(defmethod (get-next-status cell) ()
  (let ((number-live-neighbors (count-live-neighbors self)))
    (setq next-status
          (cond ((eq ':dead status)            ;empty cell
                 (if (= number-live-neighbors 3)
                     ':alive
                     ':dead))
                (t                              ;live cell
                 (if (or (= number-live-neighbors 2)
                         (= number-live-neighbors 3))
                     ':alive
                     ':dead))))))

;;; draw-outline uses message-passing (send function)
;;; to invoke the :draw-line method

(defmethod (draw-outline box) ()
  (send *game-window* :draw-line
        box-x box-y
        (+ box-x side-length) box-y)
  (send *game-window* :draw-line
        box-x box-y
        box-x (+ box-y side-length))
  (send *game-window* :draw-line
        (+ box-x side-length) box-y
        (+ box-x side-length) (+ box-y side-length))
  (send *game-window* :draw-line
        box-x (+ box-y side-length)
        (+ box-x side-length) (+ box-y side-length)))

(defmethod (draw-contents box-with-cell) ()
  (if (= (aliveness self) 1)
      ;; draw a circle to represent an alive cell
      (send *standard-output* :draw-filled-in-circle
            box-x-center box-y-center *cell-radius*)
      ;; erase a circle if the cell is dead
      (send *standard-output* :draw-filled-in-circle
            box-x-center box-y-center *cell-radius* tv:alu-andca)))
```

Using :and Method Combination

change-status illustrates the use of **:and** method combination. The generic function **change-status** has two methods written for it: one method implements **change-status** on instances of **cell**; the other method implements **change-status** on instances of **box-with-cell**. Because the **:and** type of method combination is specified in the **defgeneric** form, the handler for instances of **box-with-cell** is a combined method. The combined method first executes the method for **cell** (since **:most-specific-last** order is used). If that method returns non-**nil**, the method for **box-with-cell** is executed. This enables us to redisplay the contents of the cell only if necessary; that is, only if the status of the cell has changed.

```
(defgeneric change-status (cell-unit)
   "When applied to a cell, updates it to next-status.
    When applied to a box-with-cell, checks to see if
    the status changed.  If so, redisplays the contents."
   (:method-combination :and :most-specific-last))

;;; The following method is inherited by box-with-cell,
;;; combined with the :and type of method combination.
;;; The return value is important because it determines
;;; whether or not the box-with-cell needs to be redisplayed:

(defmethod (change-status cell) ()
  (if (eq status next-status)
      nil                           ;returns nil if no change
      (setq status next-status)))    ;returns non-nil if status changed

(defmethod (change-status box-with-cell) ()
  (draw-contents self))

;;;Note that the combined method for change-status of a box-with-cell
;;;looks like this:
;;;      (and (method for cell)
;;;           (method for box-with-cell))
```

Writing a Method for make-instance

We now write a method for **make-instance** of **box-with-cell**. This method is run every time a new instance of **box-with-cell** is made. It does some further initialization of the new instance, depending on the fact that the instance variables **box-x** and **box-y** are initialized.

```
(defmethod (make-instance box-with-cell) (&rest ignore)
  (setq box-x-center (round (+ box-x (* .5 *side-length*))))
  (setq box-y-center (round (+ box-y (* .5 *side-length*)))))
```

17.3.0.6 Using Flavors in Conjunction with an Array

Now that we have defined the flavors, generic functions, and methods for Life, we need only put the pieces together to complete the program:

```
;;; play-life-game is the top-level function that plays the Life game.

(defun play-life-game (&optional (generations 3))
  (set-up-game-board)                        ;initialize gameboard
  (iterate-game-board #'find-neighbors)
  (iterate-game-board #'draw-outline)        ;display gameboard grid
  (iterate-game-board #'draw-contents)       ;display initial set-up
  (loop for i from 1 to generations
        do
    (iterate-game-board #'get-next-status)   ;compute next-status
    (iterate-game-board #'change-status)))   ;update status & display

;;; *game-board* is a 2-dimensional array:
;;; outer border contains dead cells (easier to compute live neighbors)
;;; outer border is not displayed
;;; inner part contains box-with-cells (dead or alive by random)

(defun set-up-game-board ()
  (setq *game-board* (make-array (list (+ *number-boxes-on-axis* 2)
                                       (+ *number-boxes-on-axis* 2))))

  ;; initialize the border with dead cells
  (loop for x-pos from 0 to (1+ *number-boxes-on-axis*)
        do
      (setf (aref *game-board* x-pos 0)
            (make-instance 'cell :status ':dead))
      (setf (aref *game-board* x-pos (1+ *number-boxes-on-axis*))
            (make-instance 'cell :status ':dead)))
  (loop for y-pos from 0 to (1+ *number-boxes-on-axis*)
        do
      (setf (aref *game-board* 0 y-pos)
            (make-instance 'cell :status ':dead))
      (setf (aref *game-board* (1+ *number-boxes-on-axis*) y-pos)
            (make-instance 'cell :status ':dead)))

  ;; now initialize the inner part of the array with box-with-cells
```

```
 (loop for x-pos from 1 to *number-boxes-on-axis*      ;inner part
         for x-offset from 0 by *side-length*
       do
   (loop for y-pos from 1 to *number-boxes-on-axis*
         for y-offset from 0 by *side-length*
         do
     ;; fill up with box-with-cells
     (setf (aref *game-board* x-pos y-pos)
           (make-instance 'box-with-cell
                          :x x-pos
                          :y y-pos
                          :box-x (+ *x-top-left-corner* x-offset)
                          :box-y (+ *y-top-left-corner* y-offset))))))

;;; iterate-game-board accesses the inner part of *game-board* and
;;; applies operation to each box-with-cell.  Any operation that can be
;;; used on a single cell or box-with-cell can be used.  For example:
;;;    draw-outline, draw-contents, get-next-status, change-status

(defun iterate-game-board (operation)
  (loop for y from 1 to *number-boxes-on-axis*
        do                               ;inner part of *game-board*
    (loop for x from 1 to *number-boxes-on-axis*
          do
      (funcall operation (aref *game-board* x y)))))
```

17.3.0.7 Compiling Flavor Methods

The last line in the file that contains this program is:

```
(compile-flavor-methods)
```

This causes combined methods to be compiled at compile-time, and the data structures to be built at load-time, rather than both happening at run-time. This speeds up the program considerably the first time it is run.

17.4 Flavors Tools

Flavors stores a lot of information about defined flavors, generic functions, and methods in its internal framework. Flavors tools are designed to give you access to that information in a form useful for program development and debugging.

The tools are divided into three groups:

- Show Flavor Commands

  The Show Flavor commands were designed and implemented in Release 7.0 in response to a need for a variety of powerful tools for developing, debugging, and understanding Flavors-based programs. You can use these commands in the command processor, the editor, and the Flavor Examiner. In Dynamic windows, you can also specify them using the mouse. Click right on a displayed instance, flavor name, generic function name, or method name for a menu of Show Flavor commands.

- Zmacs Commands for Flavors

  These Zmacs commands help you deal with issues that come up when you are editing definitions of flavors, methods, and generic functions. These commands are available only in the editor.

- The Flavor Examiner

  This is an environment for using the Show Flavor commands. Use `SELECT X` for the Flavor Examiner.

17.4.1 Summary of Show Flavor Commands

The Symbolics programming environment provides the following tools for analyzing Flavors-based programs:

Show Flavor Components *flavor keywords...*
  Answers: What is the order of flavor components, and why did the system pick that order?

Show Flavor Dependents *flavor keywords...*
  Answers: What flavors inherit from this one?

Show Flavor Differences *flavor-1 flavor-2 keywords...*
  Answers: What are the differences between two flavors?

Show Flavor Functions *flavor keywords*
  Answers: What internal flavor functions are defined for this flavor?

Show Flavor Handler *operation flavor keywords...*
  Answers: When an operation (generic function or message) is applied to an instance of a given flavor, what methods implement the operation? What method combination type is used? What is the order of methods in the handler?

Show Flavor Initializations *flavor keywords...*
  Answers: How are new instances of this flavor initialized?

Show Flavor Instance Variables *flavor keywords...*
Answers: What state is maintained by instances of this flavor?

Show Flavor Methods *flavor keywords...*
Answers: What methods are defined for this flavor, or inherited from its component flavors?

Show Flavor Operations *flavor keywords...*
Answers: What operations (generic functions and messages) are supported by instances of this flavor?

Show Generic Function *operation keywords...*
Answers: What are the general characteristics of this generic function? What flavors provide a method for this generic function? What methods are implemented for this generic function?

These commands accept keywords that modify their behavior. Keyword options are available to request information in brief form, in detailed form, sorted by flavor, and so on. See the section "Keyword Options for Show Flavor Commands", page 398.

For details of each of these commands, and examples: See the section "Show Flavor Commands", page 399.

17.4.2 Entering Input for Show Flavor Commands

The Show Flavor commands are integrated with the Dynamic Lisp Listener environment.

To give a Show Flavor command you can:

- Type the name of the command in the command processor. Completion is offered for the command names.

- Highlight a flavor, instance, generic function, or method with the mouse. Now when you click right, a menu of Show Flavor commands appears.

To enter arguments to the commands you can:

- Type the desired argument. Completion is supported. You can include or omit any package prefixes.

- Point with the mouse to the desired object on the screen (such as a flavor, generic function, or method) and click left.

Note that input you have typed (such as a flavor name) is not mouse-sensitive. The output of a previous Show Flavor command is mouse-sensitive.

17.4.3 Output of Show Flavor Commands

The output of Show Flavor commands is mouse-sensitive. Sometimes the output is abbreviated. For example, a method might be represented by the name of the flavor that provides it. Even when the representation of the method is abbreviated, the method is mouse-sensitive.

You can use the output by clicking left on the representation of an object to enter it as input to another Show Flavor command, or clicking right to get a menu of commands. This way you can use the output of previous commands to continue to explore the universe of defined flavors.

17.4.4 Keyword Options for Show Flavor Commands

This section gives a general description of the keyword options accepted by most Show Flavor commands. Some of the keywords have a different meaning for each command. The documentation on the individual commands states which keywords are accepted, and describes the meaning of the keywords for that command (if they differ from the meaning described here).

17.4.4.1 Altering the Output Format

:sort	{alphabetical, flavor} Indicates how to sort the display. Alphabetical means the output is sorted alphabetically, ignoring package prefixes. For example, the output of Show Flavor Methods is sorted alphabetically by generic function name. Alphabetical is the default. If flavor is specified, the output is sorted according to the order of flavor components.

17.4.4.2 Restricting the Output

:brief	{yes, no} If yes, requests a brief answer to the question asked. The exact meaning of :brief varies for each command.
:locally	{yes, no} If yes, specifies that inherited characteristics are not to be shown.
:match	{*string*} Requests only those things (flavors, methods, generics, or whatever was requested) that match this substring. If *string* is omitted, requests all of them. :match has a different meaning for each command. This option lets you pare down the output.

17.4.4.3 Requesting Additional Output

:detailed	{yes, no} If yes, requests a detailed answer to the question asked. The exact meaning of :detailed varies for each command.

:functions	{*string*} Requests internal functions to flavors that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no internal functions are requested.
:initializations	{*string*} Requests initializations that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no initializations are requested.
:instance variables	{*string*} Requests instance variables that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no instance variables are requested.
:methods	{*string*} Requests methods for generics that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no methods are requested.

17.4.4.4 Redirecting the Output

:output destination	{buffer, stream, file, window} Redirects the output of the command to a buffer, stream, file, or window. When you enter :output destination buffer, you are prompted for the name of a buffer.

17.4.5 Show Flavor Commands

The following commands show attributes of a flavor, generic function, method, or handler. Only those keywords that are specific to each command (or have a different meaning for each command) are explained in the command descriptions. For an explanation of any keywords not covered in the command descriptions: See the section "Keyword Options for Show Flavor Commands", page 398.

17.4.5.1 Show Flavor Components Command

Show Flavor Components *flavor keywords*

Shows the order of the components of this flavor.

keywords	:brief, :detailed, :duplicates, :functions, :initializations, :instance variables, :match, :methods, and :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.
:duplicates	{yes, no} Indicates whether or not to display duplicate occurrences of flavors. The default is no.

:brief	{yes, no} yes indicates that the output should not be indented to show the structure. The default is no.

The flavor components are ordered from top to bottom. The top flavor is the *most specific flavor* in the ordering. The indentation graphically represents which flavors are components of which other flavors. In the example below, **tv:minimum-window** has six direct components: **tv:essential-expose**, **tv:essential-activate**, **tv:essential-set-edges**, **tv:essential-mouse**, **tv:essential-window**, and **flavor:vanilla**.

When you use the :duplicates keyword and show the components of complex flavors, you notice special symbols in the display. For example:

```
Command: Show Flavor Components TV:MINIMUM-WINDOW :Duplicates
    --> TV:MINIMUM-WINDOW
          TV:ESSENTIAL-EXPOSE
            [TV:ESSENTIAL-WINDOW] ↓
          TV:ESSENTIAL-ACTIVATE
            [TV:ESSENTIAL-WINDOW] ↓
          TV:ESSENTIAL-SET-EDGES
            [TV:ESSENTIAL-WINDOW] ↓
          TV:ESSENTIAL-MOUSE
          TV:ESSENTIAL-WINDOW
            TV:SHEET
              SI:OUTPUT-STREAM
                SI:STREAM
          FLAVOR:VANILLA
```

Bracketed flavors are duplicates that are included by the parent flavor here, but are not ordered in this position because of some ordering constraint. They appear in another place in the display without brackets, in their correct order. All bracketed components have an arrow beside them. A down-arrow indicates that this component's position in the ordering is later in the display. An up-arrow indicates that this component's position in the ordering is earlier in the display; these occurrences are infrequent.

For example, the flavor **tv:essential-window** is a component of four other components: **tv:essential-expose**, **tv:essential-activate**, **tv:essential-set-edges**, and **tv:minimum-window** itself. Its correct position in the ordering is directly after **tv:essential-mouse**, where it appears without brackets.

You can read the order of flavor components by reading all unbracketed flavors from top to bottom, ignoring punctuation. If :duplicates is no, this is all that is displayed.

For information on how the order is determined: See the section "Ordering Flavor Components", page 382.

17.4.5.2 Show Flavor Dependents Command

Show Flavor Dependents *flavor keywords*

Shows the names of flavors that are dependent on this flavor.

keywords	:brief, :detailed, :duplicates, :functions, :initializations, :instance variables, :levels, :match, :methods, :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.
:brief	{yes, no} yes indicates that the output should not be indented to show the structure. The default is no.
:duplicates	{yes, no} Indicates whether or not to display duplicate occurrences of flavors. The default is no.
:levels	{all, *integer*} Specifies how many levels of indirect dependency to display. The default is all, which shows all levels. For some flavors the output can be voluminous, and it is helpful to use :levels to pare it down.

A dependent flavor is a flavor that uses this flavor as a component (directly or indirectly). This is useful in program development or debugging, to answer the question "What flavors will be affected if I change the definition of this flavor?" For example:

```
Command: Show Flavor Dependents TV:SCROLL-WINDOW-WITH-DYNAMIC-TYPEOUT
    --> TV:SCROLL-WINDOW-WITH-DYNAMIC-TYPEOUT
          TV:BASIC-PEEK
            TV:PEEK-PANE
          TV:BASIC-TREE-SCROLL
            LMFS:AFSE-MIXIN
              LMFS:FSMAINT-AFSE-PANE
            LMFS:FSMAINT-HIERED-PANE
            TV:MOUSABLE-TREE-SCROLL-MIXIN
              TV:TREE-SCROLL-WINDOW
```

The output is indented to clarify which flavor is built on which component flavors. The structure of the output is the inverse of the output of Show Flavor Components. In this example, **tv:basic-peek** is a direct dependent of **tv:scroll-window-with-dynamic-typeout**, and **tv:peek-pane** is a direct dependent of **tv:basic-peek**.

17.4.5.3 Show Flavor Differences Command

Show Flavor Differences *flavor1 flavor2 keywords*

Shows the characteristics that two flavors have in common, and the characteristics in which they differ.

keywords	:match and :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.
:match	{*string*} Displays only those generic functions or messages that match the given substring.

This is most useful for two flavors that share many characteristics. Here is some sample output:

```
Command: Show Flavor Differences TV:ESSENTIAL-WINDOW TV:MINIMUM-WINDOW
     --> TV:ESSENTIAL-WINDOW and TV:MINIMUM-WINDOW have
         common components:
            flavors...
         TV:MINIMUM-WINDOW has other components:
            flavors...

         Differences in :ACTIVATE methods from TV:ESSENTIAL-WINDOW
                                              to TV:MINIMUM-WINDOW
            TV:SHEET before, primary,
            TV:ESSENTIAL-ACTIVATE after [added]

         Differences in handling of :BURY
            Flavor TV:ESSENTIAL-WINDOW does not handle :BURY
            Methods of TV:MINIMUM-WINDOW:
               TV:ESSENTIAL-EXPOSE wrapper, TV:ESSENTIAL-ACTIVATE
            more differences...
```

First, the common components are displayed. Second, the extra components of either (or both) flavors are displayed. Third, any differences in handling of generic functions are displayed.

In this example, **tv:minimum-window** has one method for **:activate** that **tv:essential-window** does not have: an **:after** method provided by flavor **tv:essential-activate**. The term `[added]` indicates that this method is defined for the second flavor but not for the first flavor. If the command had been given such that *flavor-1* was **tv:minimum-window** and *flavor-2* was **tv:essential-window**, the term would have been `[deleted]`. To interpret which flavor "adds" or "deletes" a method, look at the line that defines the perspective: `"Differences in :ACTIVATE methods from TV:ESSENTIAL-WINDOW to TV:MINIMUM-WINDOW"`.

When comparing two complex flavors, the output can be voluminous. You can use :match to pare down the output so it answers a specific question. For example:

```
Command: Show Flavor Differences DYNAMIC-LISP-LISTENER SHEET :Match
         screen
    --> information about common and different components...
        Difference in handling of :FULL-SCREEN
          Method of DW::DYNAMIC-LISP-LISTENER: TV:ESSENTIAL-SET-EDGES
          Flavor TV:SHEET does not handle generic operation :FULL-SCREEN
        another difference...
        5 local functions found with no differences:
          TV:SCREEN-MANAGE-RESTORE-AREA
          TV:SCREEN-MANAGE-CLEAR-AREA
          TV:SCREEN-MANAGE-CLEAR-UNCOVERED-AREA
          TV:SCREEN-MANAGE-CLEAR-RECTANGLE
          TV:SCREEN-MANAGE-MAYBE-BLT-RECTANGLE

        120 differing local functions were found that did not
        contain the substring "screen".
```

17.4.5.4 Show Flavor Handler Command

Show Flavor Handler *generic-function flavor keywords*

Provides information on the handler that performs *generic-function* on instances of *flavor.*

keywords	:code and :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.
:code	{yes, no, detailed} Specifies whether the Lisp code of the handler should be displayed. The default is no. Yes displays a template that resembles the actual code of the handler. Detailed displays the actual code of the handler. This displays some internal functions and data structures of the Flavors system. For most purposes, yes is more useful than detailed.

If the handler is a single method (not a combined method), its function spec is given:

```
Command: Show Flavor Handler CHANGE-STATUS CELL
    --> The handler for CHANGE-STATUS of an instance of CELL is
        the method (FLAVOR:METHOD CHANGE-STATUS CELL).
        The method-combination type is :AND :MOST-SPECIFIC-LAST.
```

If the handler is a combined method, the method combination type and order of methods are displayed. In the following example, the methods used in the combined method are represented by the names of the flavors that implement them. Even in this abbreviated format, the representation of the method is mouse-sensitive.

```
Command: Show Flavor Handler CHANGE-STATUS BOX-WITH-CELL
     --> The handler for CHANGE-STATUS of an instance of
         BOX-WITH-CELL is a combined method, with
         method-combination type :AND :MOST-SPECIFIC-LAST.
         The methods in the combined method, in order of
         execution, are: CELL, BOX-WITH-CELL
```

For combined methods, :code yes is useful. It requests a template that resembles the actual code of the handler:

```
Command: Show Flavor Handler CHANGE-STATUS BOX-WITH-CELL :Code yes
     --> The handler for CHANGE-STATUS of an instance of
         BOX-WITH-CELL is a combined method, with
         method-combination type :AND :MOST-SPECIFIC-LAST.
         (DEFUN (FLAVOR:COMBINED CHANGE-STATUS BOX-WITH-CELL)
                  (SELF SYS:SELF-MAPPING-TABLE FLAVOR::.GENERIC.
                   &REST FLAVOR::DAEMON-CALLER-ARGS.)
           (AND call (FLAVOR:METHOD CHANGE-STATUS CELL)
                call (FLAVOR:METHOD CHANGE-STATUS BOX-WITH-CELL)))
```

17.4.5.5 Show Flavor Initializations Command

Show Flavor Initializations *flavor keywords*

Shows the initialization keywords accepted by **make-instance** of this flavor, and any default initial values.

keywords :detailed, :locally, :match, :sort, and :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.

:detailed {yes, no} The default is no, which requests the allowed initialization keywords that can be given to **make-instance** of this flavor, including init keywords and initable instance variables. If :detailed is yes, any additional instance variables are also shown; these are not initable instance variables. They are initialized by default values given in the **defflavor** form. Also, any initialization methods are shown. In other words, when :detailed is no, you see the initializations from an external perspective (useful for making an instance). When :detailed is yes, you see the initializations from an internal perspective and gain information about how the flavor is constructed internally.

:locally	{yes, no} If yes, inherited initializations are not shown. The default is no, which requests all initializations defined for this flavor or inherited by this flavor.
:match	{*string*} Requests only those initializations matching the given substring.

For example:

```
Command: Show Flavor Initializations BOX-WITH-CELL :Detailed
    -->  Instances of BOX-WITH-CELL are created in the default area
         Another area can be specified with the keyword :AREA
         Initialization keywords that initialize
           instance variables:
            :BOX-X → BOX-X
            :BOX-Y → BOX-Y
            :SIDE-LENGTH → SIDE-LENGTH, default is *SIDE-LENGTH*
            :STATUS → STATUS, default is (IF (EVENP (RANDOM 2))
                                              ':ALIVE ':DEAD)
            :X → X
            :Y → Y
         Initialization method:
            MAKE-INSTANCE method: BOX-WITH-CELL
```

17.4.5.6 Show Flavor Instance Variables Command

Show Flavor Instance Variables *flavor keywords*

Shows the state maintained by instances of the given flavor.

keywords	:detailed, :locally, :match, :output destination and :sort. See the section "Keyword Options for Show Flavor Commands", page 398.
:detailed	{yes, no} If yes, the attributes of the instance variables are shown, such as their accessibility or initializations. The default is no.
:locally	{yes, no} If yes, inherited instance variables are not shown. The default is no, which shows all instance variables defined for this flavor or inherited by this flavor.
:sort	{alphabetical, flavor} If flavor, each instance variable is displayed along with the component flavor that provides it. The default is alphabetical.

For example:

```
Command: Show Flavor Instance Variables CELL
     --> NEIGHBORS
         NEXT-STATUS
         STATUS
         X
         Y
```

17.4.5.7 Show Flavor Methods Command

Show Flavor Methods *flavor*

Displays all methods defined for the given flavor.

Keywords	:locally, :match, :output destination, and :sort. See the section "Keyword Options for Show Flavor Commands", page 398.
:locally	{yes, no} If yes, inherited methods are not shown. The default is no, which shows all methods defined for this flavor or inherited by this flavor.
:match	{*string*} Requests only those methods for generic functions that match the given string.

Each line of output contains the name of the generic function, followed by the name of each flavor that provides a method for the generic function. If the method is not a primary method, its type is also displayed.

```
Command: Show Flavor Methods BOX-WITH-CELL
    --> ALIVENESS method: CELL
        CHANGE-STATUS: methods: CELL, BOX-WITH-CELL
        COUNT-LIVE-NEIGHBORS method: CELL
        :DESCRIBE method: FLAVOR:VANILLA
        others...
```

This command is similar to Show Flavor Operations. See the section "Show Flavor Operations Command", page 407. The difference between the two commands is in the perspective:

Show Flavor Methods displays information from an internal perspective, answering the question: What methods are defined for this flavor, or inherited from its component flavors?

Show Flavor Operations displays information from an external perspective, answering the question: What operations (generic functions and messages) are supported by instances of this flavor?

17.4.5.8 Show Flavor Operations Command

Show Flavor Operations *flavor keywords*

Shows all operations supported by instances of the given flavor, including generic functions and messages.

keywords	:detailed, :match, and :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.
:detailed	{yes, no} If yes, the display shows the arguments of each operation. The default is no.
:match	{*string*} Shows only those operations matching the given substring.

For example:

```
Command: Show Flavor Operations BOX-WITH-CELL
    --> ALIVENESS
        CHANGE-STATUS
        COUNT-LIVE-NEIGHBORS
        :DESCRIBE
        MAKE-INSTANCE
        SYS:PRINT-SELF (:PRINT-SELF)
        others...
```

One of the operations can be performed by using the generic function **sys:print-self** or sending the message **:print-self**. This operation was defined with **defgeneric**, using the **:compatible-message** option.

This command is similar to Show Flavor Methods. See the section "Show Flavor Methods Command", page 406. The difference between the two commands is in the perspective:

Show Flavor Operations displays information from an external perspective, answering the question: What operations (generic functions and messages) are supported by instances of this flavor?

Show Flavor Methods displays information from an internal perspective, answering the question: What methods are defined for this flavor, or inherited from its component flavors?

17.4.5.9 Show Flavor Functions Command

Show Flavor Functions *flavor keywords*

Shows internal flavor functions for the given flavor.

keywords	:locally, :match, :output destination, and :sort. See the section "Keyword Options for Show Flavor Commands", page 398.
:locally	{yes, no} If yes, inherited internal flavor functions are not shown. The default is no, which shows all internal flavor functions defined for this flavor or inherited by this flavor.
:match	{*string*} Displays only those internal functions that match the given substring.

Internal flavor functions are defined by **defun-in-flavor, defmacro-in-flavor**, and **defsubst-in-flavor**. See the section "Defining Functions Internal to Flavors", page 443.

```
Command: Show Flavor Functions TV:MAKE-WINDOW
    --> TV:ADJUST-MARGINS
        SI:ANY-TYI-CHECK-EOF
        SI:ASSURE-INSIDE-INPUT-EDITOR
        others...
```

17.4.5.10 Show Function Arguments Command

Show Function Arguments *function-spec*

Shows arguments of the function specified by *function-spec.*

17.4.5.11 Show Generic Function Command

Show Generic Function *operation keywords*

Shows information on the given *operation,* which can be a generic function or message.

keywords	:flavors, :methods and :output destination. See the section "Keyword Options for Show Flavor Commands", page 398.
:methods	{yes, no} yes displays all methods for the generic function, and their types.
:flavors	{yes, no} yes displays the flavors that implement methods for the generic function.

For example:

```
Command: Show Generic Function CHANGE-STATUS
     --> Generic function CHANGE-STATUS takes arguments: (CELL-UNIT)
         This is an explicit DEFGENERIC in file SYS:EXAMPLES;FLAVOR-LIFE.
         Method-combination type is :AND :MOST-SPECIFIC-LAST.
```

17.4.6 Summary of Zmacs Commands for Flavors, Generic Functions, and Methods

This section lists the Zmacs commands that are related to flavors, generic functions and methods. In many cases the name of the command (and the use of the HELP key) is enough to begin using the command. The details of these commands are described elsewhere in the documentation.

Any tools that give information on ordinary functions can be applied to generic functions. The following Zmacs commands work for generic functions. See the section "Finding Out About Existing Code" in *Program Development Utilities.*

Quick Arglist `c-sh-A`
Show Documentation `m-sh-D`
Long Documentation `c-sh-D`
Function Apropos (`m-X`)
List Callers (`m-X`)
Multiple List Callers (`m-X`)
Edit Callers (`m-X`)
Multiple Edit Callers (`m-X`)

The following Zmacs commands are described in detail elsewhere: See the section "Zmacs Commands for Flavors, Generic Functions, and Methods", page 410.

Edit Definition `m-.`
Show Effect of Definition (`m-X`)
Show Documentation Flavor `m-sh-F`
Kill Definition (`m-X`)
Cleanup Flavor (`m-X`)
Add Patch Cleanup Flavor (`m-X`)
Insert Cleanup Flavor Forms (`m-X`)
List Methods (`m-X`)
Edit Methods (`m-X`)
List Combined Methods (`m-X`)
Edit Combined Methods (`m-X`)

The following Zmacs commands provide the same functionality as their command processor counterparts: See the section "Show Flavor Commands", page 399.

There are two ways to enter the Zmacs Show Flavor commands: with or without a numeric argument of `c-U`. To enter a numeric argument, press `c-U m-X` before the command name. Without a numeric argument, you are prompted only for the required arguments. With a numeric argument, you are also prompted for keyword options. See the section "Keyword Options for Show Flavor Commands", page 398.

Show Flavor Components (m-X)
Show Flavor Dependents (m-X) *(see below)*
Show Flavor Differences (m-X)
Show Flavor Functions (m-X)
Show Flavor Handler (m-X)
Show Flavor Initializations (m-X)
Show Flavor Instance Variables (m-X)
Show Flavor Methods (m-X)
Show Flavor Operations (m-X)
Show Generic Function (m-X)

(m-X) Show Flavor Dependents accepts numeric arguments other than c-U; the argument specifies how many levels of indirection to display. If you enter m-2 m-X Show Flavor Dependents, two levels of indirection are displayed.

17.4.7 Zmacs Commands for Flavors, Generic Functions, and Methods

Edit Definition (m-.)

This command is one of the most valuable tools of the system. When you are developing or debugging programs, you can use m-. to find the definition of an ordinary function, generic function, flavor, method, variable, package, or other type of definition. Completion is supported on the definition, if it is already in an editor buffer.

m-. prompts for a definition to find. You can enter a large variety of representations, and m-. figures out what definition you are seeking. For example, you can enter symbols with or without package prefixes.

You can provide any of the following responses to the m-. prompt:

symbol Finds the definition of *symbol,* which can be an ordinary function or generic function. For generic functions, the **defgeneric** form is found if one exists; all existing methods are also found. *symbol* can also be one of: variable, package, **defstruct** structure, flavor, or other types of definitions.

(generic-function flavor)

Finds the definitions of one method that

	implements *generic-function* on instances of *flavor* and asks if you mean that method. If not, it proceeds to find other methods, including special-purpose methods such as **:before**, **:after**, **:default**, and so on.
(*symbol property*)	Finds the function named by function spec (*:property symbol property*). This is a handy abbreviation.
function-spec	Finds the definitions of *function-spec*. For example, you could enter **(flavor:method change-status cell)** to find the method of that function spec. Often it is more convenient to enter the list **(change-status cell)** instead.

When the requested Lisp object has multiple definitions, one of them is displayed. You can then use c-U m-. to cycle through the other definitions. Also, a list of all definitions and the files they are located in is stored in a buffer called *Definitions-*n**. The position of the cursor in that buffer controls where c-U m-. will go next.

You can also point at forms with the mouse, in a buffer or in other windows, and click m-left to edit the definition.

Show Effect of Definition (m-X)

Predicts the effect of evaluating the current definition (the definition the cursor is inside), or the current region, if a region is highlighted. **defflavor**, **defmethod**, **defgeneric**, **defwrapper**, and **defwhopper** forms are understood, among others.

Note: If the current definition is a method already defined in the world, this command predicts the effect of killing the definition of the method.

Show Documentation Flavor (m-sh-F)

Displays the documentation from the online documentation set for the current flavor. The current flavor is the flavor whose name the cursor is on, or the flavor marked by a region if a region is marked.

Kill Definition (m-X)

Removes the current definition from the editor buffer and the world. If a patch is in progress, this also removes the definition from the world being patched. The current definition is the definition that the cursor is inside, or the definition marked by a region if a region is marked.

Cleanup Flavor (m-X)

Prompts for a flavor name. It then removes from the world any methods defined for that flavor that have been removed from the editor buffer for the file where they were defined.

Add Patch Cleanup Flavor (m-X)

Prompts for a flavor name. It then inserts **fundefine** forms in the current patch for any methods defined for that flavor that have been removed from the editor buffer for the file where they were defined.

Insert Cleanup Flavor Forms (m-X)

Prompts for a flavor name. It then inserts **fundefine** forms at the current point in the buffer for any methods defined for that flavor that have been removed from the editor buffer for the file where they were defined.

List Methods (m-X)

Prompts you for a generic function name (or message name). Lists the methods of all flavors that handle the generic function, in a mouse-sensitive display. To edit one of the methods, click on its function spec in the display. Alternatively, you can use c-. to edit a method. c-. cycles through the methods, each time choosing the next method for you to edit.

Edit Methods (m-X)

Prompts you for a generic function name (or message name). One of the definitions is found and pulled into an editor buffer. c-. cycles through the methods, each time choosing the next method for you to edit.

If more than one definition is available, the list of definitions and their source files is stored in a buffer *METHODS-*n**.

List Combined Methods (m-X)

Prompts for a generic function name, then for a flavor name. It then lists the methods for the specified generic function when applied to the specified flavor. This is a mouse-sensitive display. To edit one of the methods, click on its function spec in the display. Alternatively, you can use c-. to edit a method. c-. cycles through the methods, each time choosing the next method for you to edit.

Error messages appear if the flavor does not handle the generic function, or if the flavor requested is not a composed, instantiated flavor.

List Combined Methods (m-X) can be useful for predicting what a flavor will do in response to a generic function. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them.

Edit Combined Methods (m-X)

Asks you for a generic function name and a flavor name. This command finds all the methods that are called if that generic function is called on an instance of the given flavor. You can point to the generic function and flavor with the mouse; completion is available for the flavor name. As in Edit Methods (m-X), the command skips the display and proceeds directly to the editing phase.

17.4.8 Flavor Examiner

The Flavor Examiner enables you to examine flavors, methods, generic functions, and internal flavor functions defined in the Lisp environment. It is a congenial environment for using the Show Flavor commands.

You can select the Flavor Examiner with `SELECT X`, or with the Select Activity Flavor Examiner command.

Figure 11 shows the initial window.

The Flavor Examiner window is divided into five panes.

Menu of Commands – the top-left pane

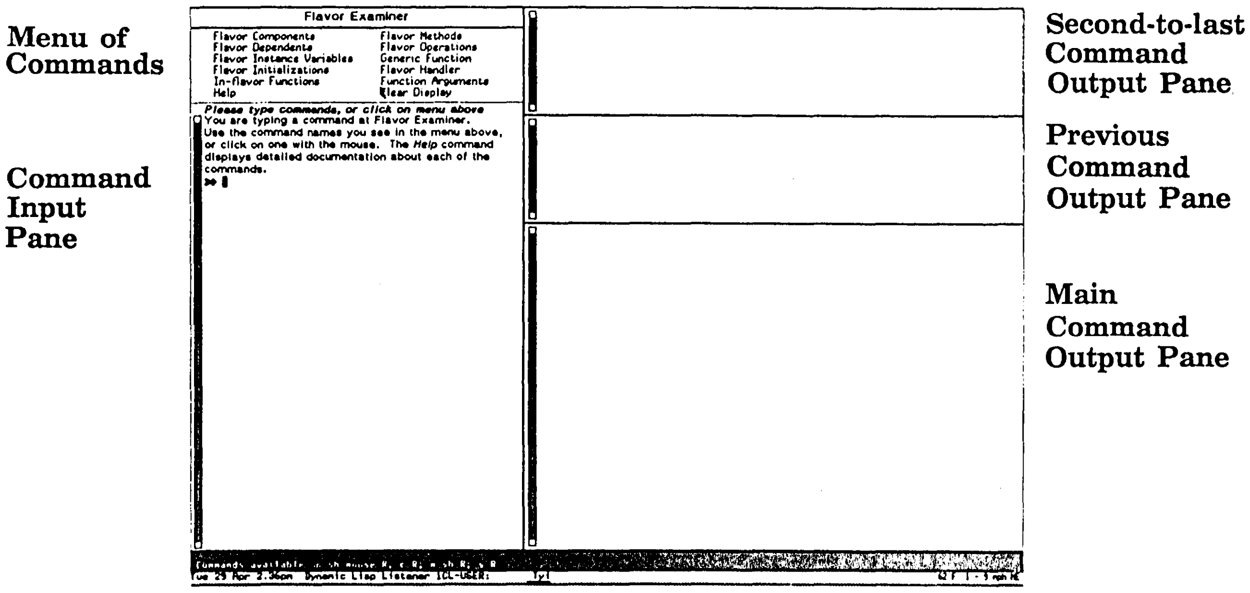
The top-left pane offers a menu of flavor-related commands, such as `Flavor Components`; this is the same as the Show Flavor Components command. You can choose one of these commands by clicking left or right. Clicking left makes the command appear in the Command Input Pane. Clicking right makes the command appear and also displays the command's arguments, in a form that you can edit.

The `HELP` command displays documentation on the flavor-related commands. The `HELP` key provides information on all the CP commands you can enter.

The Flavor Examiner offers two commands for clearing and refreshing the display. The `CLEAR DISPLAY` command clears the display from the three output panes; it first asks for confirmation. The `REFRESH DISPLAY` command displays the information on the screen again.

When you click left or right on a command name, the command appears in the Command Input Pane.

Command Input Pane – the bottom-left pane



```
Please type commands, or click on menu above
You are typing a command at Flavor Examiner.
Use the command names you see in the menu above,
or click on one with the mouse.  The Help command
displays detailed documentation about each of the
commands.
→ ▮
```

```
Commands available  m sh mouse R, c R; m sh R; s R
Tue 29 Apr 2:36pm  Dynamic Lisp Listener ICL-USER:    Tyi
```

Figure 11. Flavor Examiner Window

The bottom-left pane is a command processor window. If you click on commands in the Menu of Commands, the commands appear in this window. You can enter arguments (or commands) by typing them at the keyboard. This pane saves the history of all commands entered. You can click on the scroll bar to show different parts of the history.

You are not restricted to the commands in the Menu of Commands. You can give any command processor command.

The output of all commands appears in the Main Command Output Pane.

Main Command Output Pane – the bottom-right pane

Each command's output appears here. This pane saves the history of the output of all flavor-related commands. You can use the scroll bar to show different parts of the history.

Parts of the output of flavor-related commands are mouse-sensitive. You can make use of that by clicking on a flavor name or method name to enter it as an argument to another command.

If you give commands that are not flavor-related (such as the Show Host command), the output appears in a typeout window in the Main Command Output Pane. This kind of output is not saved in the history of this pane. The typeout window is itself a dynamic window with its own history.

When the output of the current command appears in the Main Command Output Pane, the output of the previous command is copied to the Previous Command Output Pane.

Previous Command Output Pane – the middle-right pane

This pane displays the output of the previous command. This pane does not save a history, but the second-to-last command is copied to the Second-to-Last Command Output Pane.

Second-to-last Command Output Pane – the top-right pane

This pane displays the output of the second-to-last command. This pane does not save a history. When another command is given, the contents of the Previous Command Output Pane are copied to this pane. Similarly, the contents of the Main Command Output Pane are copied to the Previous Command Output Pane.

17.5 Summary of Flavor Functions and Variables

This summary gives a brief description of all functions, macros, special forms, and variables related to Flavors. Each of the symbols listed here is described in detail elsewhere: See the section "Dictionary of Flavor Functions and Variables".

Basic Use Of Flavors

defflavor	Defines a flavor.
make-instance	Creates a new instance of a flavor.
defgeneric	Defines a generic function; enables you to specify options or documentation pertaining to a generic function as a whole.
defmethod	Defines a method that performs a generic function on objects of a given flavor.
compile-flavor-methods	Allows you to cause the combined methods of a program to be compiled at compile-time, and the data structures to be generated at load-time, rather than both happening at run-time.

Redefining Flavors, Instances, and Operations

change-instance-flavor	Changes the flavor of an instance.
recompile-flavor	Updates the internal data of the flavor and any flavors that depend on it.
flavor:remove-flavor	Removes the definition of a flavor.

flavor:rename-instance-variable	Changes the name of an instance variable, carrying the value of the old instance variable to the new for any existing instances.
flavor:transform-instance	Executes code when an instance is changed to a new flavor; thus enables you to perform initialization of the instance. Use this generic function by defining methods for it. It is not intended to be called.

Method Combination

define-simple-method-combination	Defines a new type of method combination that simply calls all the methods, passing the values they return to a given function.
define-method-combination	Enables you to declare a new type of method combination. Offers a rich declarative syntax.

The following tools are often used in **define-method-combination** forms:

flavor:call-component-method	Produces a form that calls the supplied function spec for a component method.
flavor:call-component-methods	Produces a form that invokes the supplied function or special form. Each argument to that function is a call to one of the methods in the supplied list of function specs.
flavor:multiple-value-prog2	Like **multiple-value-prog1** but returns all the values of the second form.
flavor:method-options	Extracts the method options portion of a method's function spec.

Internal Functions of Flavors

defun-in-flavor	Defines a function internal to a flavor.
defmacro-in-flavor	Defines a macro internal to a flavor.
defsubst-in-flavor	Defines a substitutable function internal to a flavor.

Wrappers and Whoppers

defwrapper	Defines a wrapper.
defwhopper	Defines a whopper.
continue-whopper	Calls the methods for the generic function that was intercepted by the whopper. This is intended for use in **defwhopper** forms.
lexpr-continue-whopper	Like **continue-whopper**, but the last argument should be a list of arguments to be passed. This is useful when the arguments to the intercepted generic function include an **&rest** argument.
defwhopper-subst	Defines a wrapper by combining the convenient syntax of **defwhopper** with the efficiency of **defwrapper**.

Variables

sys:*all-flavor-names*	A list of the names of all the flavors that have ever been created by **defflavor**.
flavor:*flavor-compile-trace-list*	A list of structures, each of which describes the compilation of a combined method into the run-time (not the compile-time) environment, in newest-first order.
self	When a generic function is called on an object, the variable **self** is automatically bound to that object.

Operations Supported by flavor:vanilla

flavor:vanilla	The flavor included in all flavors that provides default behavior. The default behavior includes methods for each of the following messages or generic functions:
:describe	The object should print a description of itself onto the ***standard-output*** stream.
sys:print-self	The object should output its printed representation to the specified stream.
:print-self	This is a compatible message for the **sys:print-self** generic function.
:send-if-handles	The object should perform the operation (whether generic function or message) if it has a method for it.
:which-operations	The object should return a list of the messages and generic functions it can handle.

:operation-handled-p	The object should return **t** if it has a handler for the specified operation, **nil** if it does not.
get-handler-for	Returns the method of the specified object for particular operation, or **nil** if the object has none.

Message-Passing

send	Sends a message to a flavor instance.
lexpr-send	Like **send**, except that the last argument should be a list. All elements of that list are passed as arguments.
send-if-handles	Sends a message to a flavor instance, if the flavor has a method defined for this message.
lexpr-send-if-handles	Like **send-if-handles**, except that the last element of arguments should be a list. All elements of that list are passed as arguments.

A Flavor's Handler for an Operation

flavor:compose-handler	Finds the methods that handle the specified generic operation on instances of the specified flavor.
flavor:compose-handler-source	Finds the methods that handle the specified generic operation on instances of the specified flavor, and finds the source code of the combined method (if any).
operation-handled-p	Returns **t** if the flavor of the given instance has a method defined for the given generic function or message; **nil** otherwise.
get-handler-for	Given an object and an operation, this function returns the object's method for the operation, or **nil** if it has none.
zl:get-flavor-handler-for	Given a flavor and an operation, this function returns the flavor's method for the operation or **nil** if it has none.

A Flavor's Default-init-plist

flavor:flavor-default-init-get	Retrieves a property from the default init-plist of the specified flavor.

flavor:flavor-default-init-putprop
Puts a property on the default-init-plist of the specified flavor.

flavor:flavor-default-init-remprop
Removes a property from the default init-plist of the specified flavor.

A Flavor's Instance Variables

symbol-value-in-instance
Allows you to read or write the value of an instance variable, or get a locative to an instance variable. This function is preferred over the next three functions.

boundp-in-instance
Returns **t** if the specified instance variable is bound in the given instance, **nil** otherwise.

zl:locate-in-instance
Returns a locative pointer to the cell inside the specified instance that holds the value of a specified instance variable.

zl:set-in-instance Allows you to write the value of an instance variable.

zl:symeval-in-instance
Allows you to read the value of an instance variable.

Getting Other Information on Flavors

flavor:find-flavor Informs you whether a given flavor is defined in the world.

flavor:flavor-allowed-init-keywords
Provides a list of all symbols that are valid init options for a given flavor.

flavor-allows-init-keyword-p
Informs you whether a given keyword is a valid init option for the specified flavor.

flavor:get-all-flavor-components
Returns a list of the components of the specified flavor, in the sorted ordering of flavor components.

Other Flavors Tools

instancep Returns **t** if the object is a flavor instance, **nil** otherwise.

flavor:describe-instance
Prints a description of an instance, including the values of its instance variables, to the ***standard-output*** stream.

flavor:print-flavor-compile-trace
Enables you to view information on the compilation of combined methods that have been compiled into the run-time environment.

:unclaimed-message
If an operation is performed on a flavor instance, and no appropriate method exists, the Flavor system checks for a method for the **:unclaimed-message** message, and invokes it if it exists.

sys:property-list-mixin
This mixin flavor provides methods that perform a set of operations on property lists of flavors.

flavor:generic Used in conjunction with the **:function** option for **defgeneric.**

sys:eval-in-instance
Evaluates a form in the lexical environment of an instance.

sys:debug-instance
Enters the debugger in the lexical environment of an instance.

17.6 Method Combination

Before reading this section, it is important to understand the usual way the flavor system constructs a single handler from a set of available methods for a generic function. See the section "Inheritance of Methods", page 377.

This section describes how to use different types of method combination. The system offers several built-in types of method combination. You can also define your own type of method combination, using **define-simple-method-combination** or **define-method-combination.**

To use either a built-in type or a new type of method combination that you have defined, you supply the **:method-combination** option to **defflavor** or **defgeneric.** Often you also write special-purpose methods to be used in conjunction with the type of method combination, by supplying the *options* argument to **defmethod.**

17.6.1 Using The :method-combination Option

This section describes the syntax and use of the **:method-combination** option, which can be given to **defflavor** and **defgeneric.** Each generic function has an associated type of method combination. If no method combination is explicitly specified, the default is **(:method-combination :daemon :most-specific-first).**

The syntax of this option is different for **defflavor** and **defgeneric.**

The **:method-combination** option to **defflavor** is given as follows:

```
(:method-combination
     generic-function name
     generic-function (name args...)
     ...)
```

The **:method-combination** option to **defgeneric** is given as follows:

```
(:method-combination name args...)
```

Each generic function is associated with a certain type of method combination specified by *name*. You can supply a built-in type of method combination, or a new type you have defined yourself. See the section "Built-in Types of Method Combination", page 423.

Sometimes the method combination type requires additional arguments, which are supplied as *args*.

In the built-in types of method combination, the first argument is usually the order that methods should be combined, either **:most-specific-first** or **:most-specific-last**. Often this argument is optional, and **:most-specific-first** is the default; this depends on the type of method combination used. See the section "**:most-specific-first** And **:most-specific-last** Method Order", page 422.

The following example uses **:daemon-with-or** method combination type for the generic function **fast-hardcopy**; the **:most-specific-first** method order is supplied. The **:case** method combination type is used for the generic function **set-attribute**.

```
(defflavor line-printer ((name "Tortoise")
                         (baud-rate 1200))
           ()
  :initable-instance-variables
  (:method-combination
        fast-hardcopy (:daemon-with-or :most-specific-first)
        set-attribute :case))
```

The following is an example of providing the **:method-combination** option to **defgeneric**:

```
(defgeneric hardcopy-file (device filename)
  (:method-combination :and :most-specific-first))
```

If **:method-combination** is specified for the same generic function by both **defflavor** and **defgeneric**, they must agree.

Any component of a flavor can specify the type of method combination to be used for a generic function. If more than one component of a flavor specifies a type of method combination, they must agree on the specification. Otherwise an error is signalled.

17.6.2 :most-specific-first And :most-specific-last Method Order

This section describes the meaning of **:most-specific-first** and **:most-specific-last,** two keywords often supplied as the *order* argument to the **:method-combination** option to **defflavor** and **defgeneric.**

In this example, the ordering of flavor components for **person** is:

```
(person primate living-thing flavor:vanilla)
```

person is the most specific flavor, and **flavor:vanilla** is the least specific flavor in the flavor ordering. **flavor:vanilla** is always the least specific flavor unless it is explicitly excluded from the flavor by using the **:no-vanilla-flavor** option to **defflavor.**

Several primary methods are available for the generic function **get-nutrition**:

- Flavor **person** supplies a method for **get-nutrition,** using a fork and spoon.
- Flavor **primate** supplies a method for **get-nutrition,** using hands (or paws) and teeth.
- Flavor **living-thing** supplies a very general method for **get-nutrition** that would be appropriate for any living thing, including plants.
- **flavor:vanilla** supplies no methods for **get-nutrition.**

17.6.2.1 Choosing a Single Primary Method

Some method combination types use the *order* argument to choose a single primary method from the set of available primary methods. **:daemon** method combination type is one example. If **:most-specific-first** order is used, the primary method chosen is the primary method that is supplied by the first flavor in the ordering of flavor components.

Using **:most-specific-first** method order, the method implemented by flavor **person** would be chosen.

Using **:most-specific-last** method order, the method implemented by flavor **living-thing** would be chosen.

17.6.2.2 Indicating the Order of Methods in a Combined Method

Most built-in types of method combination use the *order* argument to determine what order the methods are run in the combined method. For example, consider the **:or** type of method combination, which executes every method inside an **or** special form.

Using **:most-specific-first** order, the combined method resembles:

```
(or (method-for-person)
    (method-for-primate)
    (method-for-living-thing))
```

Using **:most-specific-last** order, the combined method resembles:

```
(or (method-for-living-thing)
    (method-for-primate)
    (method-for-person))
```

17.6.3 Built-In Types of Method Combination

You can use the following types of method combination by supplying one of the following keywords as the *name* argument to the **:method-combination** option to either **defgeneric** or **defflavor**.

The default method combination type **:daemon** is presented first. The other types appear in alphabetical order.

:daemon &optional (*order* **':most-specific-first**)

This is the default type of method combination. **:daemon** selects all **:before** and **:after** methods, and a single primary method, which is the first method in the given *order*. The combined method calls the **:before** methods in **:most-specific-first** order, then the chosen primary method, then the **:after** methods in **:most-specific-last** order. The result of the combined method is whatever the primary method returns. The combined method resembles:

```
(flavor:multiple-value-prog2
   (progn (before-method-1)
          (before-method-2))
   (primary-method-1)
   (progn (after-method-2)
          (after-method-1)))
```

:and &optional (*order* **':most-specific-first**)

Selects all primary and **:and** methods and calls them in the given *order* inside an **and** special form. If a method returns **nil**, no more methods are called. The value returned is the value of the last method called, or **nil** if any method returns **nil**. Each **:and** method can return a single value only. The combined method resembles:

```
(and (method-1)
     (method-2)
     (method-3))
```

:append &optional (*order* **':most-specific-first**)

Selects all primary and **:append** methods and calls them in the given *order* inside a call to the **append** function. Each of the methods must return a single value only, which is a list. The final result is the result of appending all these lists. The combined method resembles:

```
(append (method-1)
        (method-2)
        (method-3))
```

:case Invokes methods inside a **case** special form, using the second argument to the generic function to select the appropriate case. This argument is usually a keyword. Methods used with **:case** method combination must always include an *option* in their name, which specifies which case they implement. For example:

```
(defmethod
   (proceed subscript-out-of-bounds :new-subscript)
   (&optional (sub (prompt-and-read
                    :integer
                    "Subscript to use instead:")))
   "Supply a replacement subscript"
   (values :new-subscript sub))
```

This method is invoked by **(proceed condition :new-subscript)**, which prompts the user for a new subscript, and by **(proceed condition :new-subscript n)**, which uses **n** as the new subscript.

There are three special values of *option*, used to define special-purpose methods:

:otherwise This method is invoked if the second argument to the generic function is one for which a method has not been defined. The **:otherwise** method receives the unmatched second argument keyword as its first argument and the third and following arguments to the generic function as its remaining arguments. If no **:otherwise** method is defined for this flavor, a system-supplied method is called; it signals an error.

:which-operations
This method is invoked if the second argument to the generic function is the symbol **:which-operations**. It is expected to take no additional arguments and to return two values: a list of the supported cases, and **t** if there is an otherwise method, or **nil** if there is not. If no **:which-operations** method is defined for this flavor, a system-supplied method is called.

:case-documentation

This method is invoked if the second argument to the generic function is the symbol **:case-documentation.** It is expected to take one argument (the third argument to the generic function), which is the name of one of the cases, and is expected to return a string, which is the documentation of that case, or **nil** if no documentation is known. If no **:case-documentation** method is defined for this flavor, a system-supplied method is called; it looks for documentation strings in the methods.

Note that the usual check for consistency of **defmethod** arguments with the arguments in the **defgeneric** form is not performed when **:case** method combination type is used, since the arguments can be different for each case.

Here is an example of defining a generic function that uses **:case** method combination type:

```
(defgeneric zzbuffer-next-unlinked-line (buffer selector line)
  "for buffers with disconnected sections, crosses a hard
   section boundary."
  (:method-combination :case)
  (:method (zznode :otherwise)
           (ignore selector line) nil)  ; no unlinked lines
  (:method (zznode :foo)
           (ignore line) nil)           ; no selector here
  )
```

:daemon-with-and &optional (*order* **':most-specific-first**)

Selects all **:before**, **:after**, and **:and** methods, and a single primary method. This is like the **:daemon** method combination type, except that the primary method is wrapped in an **and** special form after the **:and** methods. To write an **:and** method, supply the keyword **:and** as the *options* argument to **defmethod**. The result is the values returned by the primary method if it is run, **nil** otherwise. The combined method resembles:

```
(flavor:multiple-value-prog2
   (progn (before-method-1)
          (before-method-2)
   (and (and-method-1)
        (and-method-2)
        (primary-method-1))
   (progn (after-method-2)
          (after-method-1)
```

The *order* argument indicates the order of **:and** methods and the choice of the primary method.

:daemon-with-or &optional (*order* **':most-specific-first**)

Selects all **:before**, **:after**, and **:or** methods, and a single primary method. This is like the **:daemon** method combination type, except that the primary method is wrapped in an **or** special form with all the **:or** methods. To write an **:or** method, supply the keyword **:or** as the *options* argument to **defmethod**. The result is either the single value returned by the first **:or** method to return non-**nil**, or the values returned by the primary method (if it is run). The combined method resembles:

```
(flavor:multiple-value-prog2
   (progn (before-method-1)
          (before-method-2)
   (or (or-method-1)
       (or-method-2)
       (primary-method-1))
   (progn (after-method-2)
          (after-method-1)
```

The *order* argument indicates the order of **:or** methods and the choice of the primary method.

This is primarily useful for flavors in which a mixin introduces an alternative to the primary method. Each **:or** method gets a chance to run before the primary method and to decide whether or not the primary method should be run; if any **:or** method returns a non-**nil** value, the primary method is not run (nor are the rest of the **:or** methods).

:daemon-with-override &optional (*order* **':most-specific-first**)

Selects all **:before**, **:after**, and **:override** methods, and a single primary method. This is similar to the **:daemon** and **:daemon-with-or** method combination types. The **:before** methods, the primary method, and the **:after** methods are run only if all of the **:override** methods return **nil**. The result is either the single value returned by the first **:override** method to return non-**nil**, or the values returned by the primary method (if it is run). To write an **:override** method, supply the keyword **:override** as the *options* argument to **defmethod**. The combined method resembles:

```
(or (override-method-1)
    (override-method-2)
    (flavor:multiple-value-prog2
       (progn (before-method-1)
              (before-method-2))
       (primary-method-1)
       (progn (after-method-2)
              (after-method-1))))
```

The *order* argument indicates the order of **:override** methods and the choice of the primary method.

:inverse-list

Selects all primary and **:inverse-list** methods. The combined method calls each method in **:most-specific-first** with one argument; these arguments are successive elements of the list given as an argument to the generic function. Returns no particular value. If the result of a **:list**-combined generic function is sent back with an **:inverse-list**-combined generic function, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

The generic function is called as follows:

(generic-function object arg1 arg2 arg3)

The combined method resembles:

```
(progn (method-1 arg1)
       (method-2 arg2)
       (method-3 arg3))
```

:list &optional (*order* '**:most-specific-first**)

Selects primary and **:list** methods and calls them in the given *order order* inside a call to the **list** function. Each method used with **:list** method combination can return a single value only. The result is a list of the returned values of the methods. The combined method resembles:

```
(list (method-1)
      (method-2)
      (method-3))
```

:max &optional (*order* '**:most-specific-first**)

Selects all primary and **:max** methods and calls them in the given *order* inside a call to the **max** function. Each method should return a numeric value. The result is the largest value of the set of values returned by the methods. The combined method resembles:

```
(max (method-1)
     (method-2)
     (method-3))
```

:min &optional (*order* **':most-specific-first)**

Selects all primary and **:min** methods and calls them in the given *order* inside a call to the **min** function. Each method should return a numeric value. The result is the smallest value of the set of values returned by the methods. The combined method resembles:

```
(min (method-1)
     (method-2)
     (method-3))
```

:nconc &optional (*order* **':most-specific-first)**

Selects all primary and **:nconc** methods and calls them in the given *order* inside a call to the **nconc** function. Each of the methods must return a single value only, which is a list. The final result is the result of concatenating these lists destructively. The combined method resembles:

```
(nconc (method-1)
       (method-2)
       (method-3))
```

:or &optional (*order* **':most-specific-first)**

Selects all primary and **:or** methods and calls them in the given *order* inside an **or** special form. If a method returns a non-**nil** value, that value is returned and none of the other methods is called; otherwise, the next method is called. Thus each method is given a chance to handle the generic function. Methods that do not intend to handle the generic function should return **nil** to give the next method a chance to try. Each method can return a single value only. The combined method resembles:

```
(or (method-1)
    (method-2)
    (method-3))
```

:pass-on *order* **&rest** *arglist*

Selects all primary and **:pass-on** methods and calls them in the given *order*. The arguments to each method are the values returned by the preceding method. The values returned by the combined method are those of the outermost call. *arglist* is the argument list, which can include the **&aux** and **&rest** keywords.

For example, the flavors, generic function, and methods are defined as follows:

```
(defflavor f1 ()())
(defflavor f2 ()(f1))
(defflavor f3 ()(f2))

(defgeneric foo (fl a b c)
  (:method-combination :pass-on
   :most-specific-first a b c))

(defmethod (foo f1) (a b c) (values a b c))
(defmethod (foo f2) (a b c) (values a b c))
(defmethod (foo f3) (a b c) (values a b c))
```

The combined method for the generic function **foo** for flavor **f3** resembles:

```
(multiple-value-setq (A B C)
    (multiple-value-setq (A B C)
        (multiple-value-setq (A B C)
            (method-1 (A B C)
          (method-2 (A B C)
      (method-3 (A B C)))))))
```

:progn &optional (*order* **':most-specific-first**)
Selects primary and **:progn** methods and calls them in the given *order* inside a **progn** special form. The result of the combined method is whatever the last method returns. The combined method resembles:

```
(progn (method-1)
       (method-2)
       (method-3))
```

:sum &optional (*order* **':most-specific-first**)
Selects all primary and **:sum** methods and calls them in the given *order* inside a call to the + function. The combined method resembles:

```
(+ (method-1)
   (method-2)
   (method-3))
```

:two-pass &optional (*order* **':most-specific-first**)
Selects all primary methods and all **:after** methods. The combined method calls the primary methods in the given *order*, then the **:after** methods in **:most-specific-last** order. Returns the values returned by the last primary method that is executed. The combined method resembles:

```
(multiple-value-prog1
   (progn (primary-method-1)
          (primary-method-2)
          (primary-method-3))
   (progn (after-method-3)
          (after-method-2)
          (after-method-1)))
```

This is the type of method combination used by the **make-instance** generic function. For more information: See the section "Writing Methods For **make-instance**", page 365.

17.6.4 Defining Special-Purpose Methods

This section describes how to define methods that are not primary methods; these are methods used in conjunction with different types of method combination. **:before** and **:after** methods are examples of special-purpose methods; they are used with **:daemon** method combination type. For examples of defining **:before** and **:after** methods: See the section "Writing Before and After-Daemons", page 360.

For information on which types or methods are appropriate for the various built-in method combination types: See the section "Types of Methods Used by Built-in Method Combination Types", page 431.

The syntax of **defmethod** is as follows:

(defmethod *(generic-function flavor options..) (arg1 arg2..) body..*)

The *options* argument is used when you are defining a special-purpose method. To define a primary method (the most common type of method), you do not supply the *options* argument.

To define an **:override** method (to be used with **:daemon-with-override** method combination):

```
(defmethod (update-display window-pane :override) (arg1 arg2..)
    body..)
```

To define a method to be used with **:case** method combination:

```
(defmethod (proceed subscript-out-of-bounds :new-subscript)
           (&optional
              (sub (prompt-and-read :integer
                                    "Subscript to use instead:")))
   "Supply a replacement subscript"
   (values :new-subscript sub))
```

17.6.4.1 Using :default Methods

A **:default** method is ignored if any primary methods are present. However, if no primary methods are defined, the **:default** method acts as a primary method.

This is useful in some types of method combination, such as **:and** and **:or**. That is, if no other methods are available to be combined in the **and** or **or** special form, the **:default** method is executed. The **:default** method might print a warning, or otherwise deal with the situation. If one or more primary methods are available, the **:default** method is not part of the combined method, and is not executed.

In the past, programmers defined **:default** methods for the most basic flavor. This usage guaranteed that the method would not be used in the combined method if a primary method were defined for any other flavor component. This was sometimes necessary, to compensate for the way that flavor components were ordered, in previous versions of the flavor system. It is no longer necessary to use **:default** methods for this purpose, because flavor component ordering works more predictably now than in previous releases.

17.6.5 Types of Methods Used by Built-in Method Combination Types

This section shows which types of methods are appropriate to be used in conjunction with each of the built-in types of method combination.

Some types of method combination expect a certain type of method to be used with them. For example, **:before** methods are used by **:daemon**, **:daemon-with-override**, **:daemon-with-or**, and **:daemon-with-and** method combination types. However, the other built-in types of method combination do not use **:before** methods.

It is important to realize that each method combination type ignores methods that are not appropriate for it. For example, if you have written a **:before** method for a generic function that uses **:list** method combination, that **:before** method will not be used in the combined method. A warning to that effect is printed.

Method Combination	*Accepted Method Types*
:and	primary, **:and**, **:default**
:append	primary, **:append**, **:default**
:case	:*keyword*, **:which-operations**, **:documentation**, **:otherwise**
:daemon	primary, **:before**, **:after**, **:default**

:daemon-with-and	primary, **:before**, **:after**, **:and**, **:default**
:daemon-with-or	primary, **:before**, **:after**, **:or**, **:default**
:daemon-with-override	primary, **:before**, **:after**, **:override**, **:default**
:inverse-list	primary, **:inverse-list**, **:default**
:list	primary, **:list**, **:default**
:max	primary, **:max**, **:default**
:min	primary, **:min**, **:default**
:nconc	primary, **:nconc**, **:default**
:sum	primary, **:sum**, **:default**
:or	primary, **:or**, **:default**
:pass-on	primary, **:pass-on**, **:default**
:progn	primary, **:progn**, **:default**
:two-pass	primary, **:after**, **:default**

The chart above shows that many types of method combination accept both primary methods and another kind of method. For example, **:and** method combination accepts primary and **:and** methods, and treats them the same when constructing the combined method. The only reason to use **:and** in the **defmethod** form is for documentation purposes; it clarifies that the method is intended to be used with **:and** method combination type. Other method types that fall into this category include: **:append**, **:or** when used with **:or** method combination, **:list**, **:max**, **:min**, **:inverse-list**, **:pass-on**, and **:progn**.

17.6.6 Example of Using Method Combination

This example might be part of a game in which each player has a country named by a color. Each country consists of a military presence, represented by the flavor **the-military**. Some countries have organized their military presence into branches represented by the flavors **army**, **navy**, and so on. In our example, **army** is built on **the-military**.

The game consists of moving armies and navies geographically. An important element of the game is keeping track of supplies, such as food and fuel. In this example, fuel is stored in three places:

- In a reserve storehouse -- managed by **the-military** (it can be used by any branch of the military that needs it)
- In an auxiliary storehouse -- managed by each branch
- In the field -- managed by each branch

The following two **defflavor** forms represent **the-military** and **army**:

```
;;; the-military keeps its own storehouse of fuel supplies
;;; the-military has supplies of diesel, gas, and coal
(defflavor the-military ((reserve-diesel-supply 1200)
                         (reserve-gas-supply 1350)
                         (reserve-coal-supply 5600))
           ())

;;; the army's auxiliary supply augments fuel stored in the field
;;; the army has supplies of diesel and gas, but no coal
(defflavor army ((aux-diesel-supply 100)
                 (field-diesel-supply 150)
                 (aux-gas-supply 225)
                 (field-gas-supply 230))
           (the-military))           ;the-military is a component
```

Each player must keep track of fuel supplies. We organize this by writing a generic function **total-fuel-supply**. This function takes two arguments: *military-group* and *fuel-type*. It returns the total available supply of the given *fuel-type* (which can be **:diesel, :coal,** or **:gas**) for that military-group (it could be an instance of **army** or **the-military**). The generic function makes use of the **:sum** type of method combination.

```
;;; define a new type of method combination called :sum
;;; :sum creates combined methods like
;;;    (+ (method-1)
;;;       (method-2))
(define-simple-method-combination :sum + t)

(defgeneric total-fuel-supply (military-group fuel-type)
  "Returns today's total supply
   of the given type of fuel available to this military group."
  (:method-combination :sum :base-flavor-last);
```

```
(defmethod (total-fuel-supply army) (fuel-type)
  (case fuel-type
        (:diesel (+ aux-diesel-supply field-diesel-supply))
        (:gas (+ aux-gas-supply field-gas-supply))
        (:otherwise 0)))

(defmethod (total-fuel-supply the-military) (fuel-type)
  (case fuel-type
        (:diesel reserve-diesel-supply)
        (:gas reserve-gas-supply)
        (:coal reserve-coal-supply)
        (:otherwise 0)))
```

When we use **total-fuel-supply** on an instance of **the-military**, it responds with the amount of fuel available in the reserve storehouse managed by the-military. The handler is just the method for **the-military**.

When we use **total-fuel-supply** on an instance of **army**, it responds with the sum total of that type of fuel available to that object of **army** flavor, including its own auxiliary storehouse, its fuel in the field, and the fuel available to it stored in the reserve storehouse of the-military as a whole. The handler is a combined method that resembles:

```
(+ (method-for-army)
   (method-for-the-military))
```

Here is an illustration of using **total-fuel-supply**:

```
(setq blue-army (make-instance 'army))
-->#<ARMY 36431263>

(total-fuel-supply blue-army :coal)
-->5600

(total-fuel-supply blue-army :diesel)
-->1450
```

17.6.7 Defining a New Type of Method Combination

This section describes the tools that enable you to define a new type of method combination.

17.6.7.1 Summary of Method Combination Functions

define-simple-method-combination
Defines a new type of method combination that simply calls all the methods, passing the values they return to a given function.

define-method-combination
Enables you to declare a new type of method combination. Offers a rich declarative syntax.

The following tools are often used in **define-method-combination** forms:

flavor:call-component-method
Produces a form that calls the supplied function spec for a component method.

flavor:call-component-methods
Produces a form that invokes the supplied function or special form. Each argument to that function is a call to one of the methods in the supplied list of function specs.

flavor:multiple-value-prog2
Like **multiple-value-prog1** but returns all the values of the second form.

flavor:method-options
Extracts the method options portion of a method's function spec.

define-simple-method-combination *name operator* &optional *single-arg-is-value pretty-name* *Special Form*

Defines a new type of method combination that simply calls all the methods, passing the values they return to the function named *operator.*

It is also legal for *operator* to be the name of a special form. In this case, each subform is a call to a method. It is legal to use a lambda expression as *operator.*

name is the name of the method-combination type to be defined. It takes one optional parameter, the order of methods. The order can be either **:most-specific-first** (the default) or **:most-specific-last**.

When you use a new type of method combination defined by **define-simple-method-combination**, you can give the argument **:most-specific-first** or **:most-specific-last** to override the order that this type of method combination uses by default.

If *single-arg-is-value* is specified and not **nil**, and if there is exactly one method, it is called directly and *operator* is not called. For example, *single-arg-is-value* makes sense when *operator* is +.

pretty-name is a string that describes how to print method names concisely. It defaults to (**string-downcase** *name*).

Most of the simple types of built-in method combination are defined with **define-simple-method-combination**. For example:

```
(define-simple-method-combination :and and t)
(define-simple-method-combination :or or t)
(define-simple-method-combination :list list)
(define-simple-method-combination :progn progn t)
(define-simple-method-combination :append append t)
```

define-method-combination *name parameters method-patterns options... body..* *Special Form*

Provides a rich declarative syntax for defining new types of method combination. This is more flexible and powerful than **define-simple-method-combination.**

name is a symbol that is the name of the new method combination type. *parameters* resembles the parameter list of a **defmacro**; it is matched against the parameters specified in the **:method-combination** option to **defgeneric** or **defflavor.**

method-patterns is a list of method pattern specifications. Each method pattern selects some subset of the available methods and binds a variable to a list of the function specs for these methods. Two of the method patterns select only a single method and bind the variable to the chosen method's function spec if a method is found and otherwise to **nil**. The variables bound by method patterns are lexically available while executing the *body* forms. See the section "*Method-patterns* Option To **define-method-combination**", page 439.

Each *option* is a list whose **car** is a keyword. These can be inserted in front of the body forms to select special options. See the section "Options Available In **define-method-combination**", page 441.

The *body* forms are evaluated to produce the body of a combined method. Thus the body forms of **define-method-combination** resemble the body forms of **defmacro**. Backquote is used in the same way. The *body* forms of **define-method-combination** usually produce a form that includes invocations of **flavor:call-component-method** and/or **flavor:call-component-methods**. These functions hide the implementation-dependent details of the calling of component methods by the combined method.

Flavors performs some optimizations on the combined method body. This makes it possible to write the body forms in a simple and easy-to-understand style, without being concerned about the efficiency of the generated code. For example, if a combined method chooses a single method and calls it and does nothing else, Flavors implements the called method as the handler rather than constructing a combined method. Flavors removes redundant invocations of **progn** and **multiple-value-prog1** and performs similar optimizations.

The variables **flavor:generic** and **flavor:flavor** are lexically available to the body forms. The values of both variables are symbols:

flavor:generic	value is the name of the generic operation whose handler is being computed.
flavor:flavor	value is the name of the flavor.

The *body* forms are permitted to **setq** the variables defined by the *method-patterns*, if further filtering of the available methods is required, beyond the filtering provided by the built-in filters of the *method-patterns* mechanism. It is rarely necessary to resort to this. Flavors assumes that the values of the variables defined by the method patterns (after evaluating the body forms) reflect the actual methods that will be called by the combined method body.

body forms must not signal errors. Signalling an error (such as a complaint about one of the available methods) would interfere with the use of flavor examining tools, which call the user-supplied method combination routine to study the structure of the erroneous flavor. If it is absolutely necessary to signal an error, the variable **flavor:error-p** is lexically available to the body forms; its value must be obeyed. If **nil**, errors should be ignored.

The syntax of **define-method-combination** is complex. We present examples here and continue with the syntax later on in this section.

17.6.7.2 Examples Of define-method-combination

This form defines the **:daemon** method-combination type:

```
(define-method-combination :daemon
                              (&optional (order ':most-specific-first))
           ;; select methods and bind them to variables
           ((before "before" :every :most-specific-first (:before))
            ;; select primary method,
            ;; if none is present, select :default method
            (primary "primary" :first order () :default)
            (after "after" :every :most-specific-last (:after)))
  ;;return values from primary method
  '(flavor:multiple-value-prog2
     ,(flavor:call-component-methods before)
     ,(flavor:call-component-method primary)
     ,(flavor:call-component-methods after)))
```

This form defines the **:two-pass** method combination type:

```
(define-method-combination :two-pass
                           (&optional (order ':most-specific-first))
          ((first "first-pass" :every order () :default)
           (second "after" :every :most-specific-last (:after)))
  ;;return values from last primary method to run
  '(multiple-value-prog1
     ,(flavor:call-component-methods first)
     ,(flavor:call-component-methods second)))
```

This form defines the **:inverse-list** method combination type:

```
(define-method-combination :inverse-list ()       ;take no parameters
        ;; select methods of type :inverse-list or :default
        ((methods "inverse-list" :every :most-specific-first
                                 () (:inverse-list) :default))
  (:arglist ignore list)
  (:method-transformer
    ;; each method receives a single argument, regardless
    (:generic-method-arglist '(list-element)))

  '(let ((list ,list))
     ,@(loop for (method . rest) on methods
           collect (flavor:call-component-method method
                    :arglist '(,(first list)))
           when rest
             collect '(setq list (cdr list)))))
```

This form defines the **:case** method combination type:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: FLAVOR -*-

(define-method-combination :case (&optional
                                   (order ':most-specific-first))
        ((case-documentation "case-documentation" :first order
                                                  (:case-documentation))
         (which-operations "which-ops" :first order (:which-operations))
         (otherwise "otherwise" :first order (:otherwise))
         (cases "case ~s" :remove-duplicates order (*)))
  (:order cases case-documentation which-operations otherwise)
  (:arglist ignore subtype &rest args)
  (:method-transformer
    ;; the cases methods receive funny arguments, and the arguments
    ;; may be different for each case, so don't bother validating them.
    (:inhibit-checking t))
(let ((alist (loop for case in cases
                   collect (list (first (method-options case)) case))))
  '(case ,subtype
     ,@(loop for (subtype method) in alist
             collect '((,subtype)
                       ,(call-component-method method :apply args)))
     ((:case-documentation)
      ,(if case-documentation
           (call-component-method case-documentation :apply args)
           '(get-case-documentation (first ,args) ',alist)))
     ((:which-operations)
      ,(if which-operations
           (call-component-method which-operations :apply args)
           '(values ',(mapcar #'car alist) ',(not (null otherwise)))))
     (otherwise
      ,(if otherwise
           (call-component-method otherwise)
           '(case-method-combination-missing-method self ',generic
                                                         ,subtype
                                                         ,args))))))
```

17.6.7.3 *Method-patterns* Option To define-method-combination

Each *method-pattern* is a list of the form:

(*variable printer filter order pattern pattern...*)

variable is a variable to be bound to the list of method function specs resulting from the application of this *method-pattern* (or to a single function spec or **nil**, in the case where *filter* is one of the symbols **:first** or **:last**). Each *pattern* selects some subset of the methods (multiple patterns are combined with **or**). The *order* specifies the ordering of this subset and the *filter* specifies further pruning of it.

printer controls the concise printing of method function specs for Flavors tools. *printer* should concisely describe the options of the function spec; the generic function name and flavor are printed separately. When Flavors tools are used, *printer* is printed on ***standard-output***. Thus *printer* can be a string, which is given to **format** along with arguments that are the elements of the **flavor:method-options** of the method's function spec. See the function **flavor:method-options** in *Symbolics Common Lisp: Language Dictionary*. Alternatively, *printer* can be a function to be called with *method-combination* and *function-spec* as its arguments. *printer* is not evaluated. The printing of **:default** methods is handled automatically, independent of *printer*.

filter is not evaluated; the filter type cannot be variable at run time. *filter* must be one of the following symbols:

:first	Selects the first method in the specified order (**nil** if there are no methods).
:last	Selects the last method in the specified order (**nil** if there are no methods).
:every	Selects all the methods.
:remove-duplicates	Selects all of the methods, excluding any duplicate methods. If duplicates are present, only the first method is selected. Duplicate methods are detected by applying **equal** to their **defmethod** *options*.

Additional filter types might be added in the future. You can also do your own filtering in the body by using **setq** on a variable.

order is a form that must evaluate to **:most-specific-first** or **:most-specific-last**. Often it is simply one of those keywords, a self-evaluating constant. Another common practice is for *order* to be one of the variables in *parameters*.

Each *pattern* is a list or (). A method matches the pattern if its *options* in the **defmethod** form match the given *pattern*. The pattern () matches methods with no *options*; that is, primary methods. If you do not specify any patterns, a default pattern of () is assumed. *pattern*s may be dotted lists.

Match is by **equal**, except that * matches anything. Each * in a pattern matches anything in that position of a method function spec. Dotted * might be useful for variable length.

You can intermix special symbols with the patterns. The only such symbol currently allowed is:

:default	If the other patterns find any methods, this is ignored. If no other methods are found, the methods matched by the pattern **(:default)** are selected.

method-patterns are considered sequentially in the order they are written. The first *method-pattern* that matches a component method with its *patterns* takes care of that method, either including it or rejecting it depending on its *filter* and any **:default** processing specified. Subsequent *method-patterns* do not see that method. This means you should put the most general patterns last if more than one *method-pattern* clause could match the same method. The methods are expected to be called in the order the method patterns are written. If this is not so, you should include an **:order** clause. See the section "Options Available In **define-method-combination**", page 441.

Any methods not taken care of by any method patterns are extraneous, and Flavors warns about them. Flavors automatically takes care of special methods, such as wrappers and whoppers; you need not do anything to handle them when defining a new method combination type.

Here is an example of the *method-patterns* used by **:case** method combination type:

```
((case-documentation "case-documentation" :first order (:case-documentation))
 (which-operations "which-operations" :first order (:which-operations))
 (otherwise "otherwise" :first order (:otherwise))
 (cases "case ~s" :remove-duplicates order (*)))
```

17.6.7.4 Options Available in define-method-combination

The *options* to **define-method-combination** include:

(**:arglist** *args...*)
: Specifies the argument list of the generic function, including the first or **self** argument. You can use **ignore** for arguments of no interest to the method-combination function, such as the **self** argument. As the *body* forms are being executed, each variable in *args* is bound to a form that accesses the relevant argument when evaluated as part of the combined-method body returned by the *body* forms. &optional and &rest are permitted in *args*. If **:dispatch** is used in the **defgeneric** form, the **:arglist** option mentions the arguments in the order received by the methods, with the dispatching argument moved to the front.

  The **:arglist** option is useful when the action of the combined-method depends on the arguments (rather than simply passing the arguments on to the component methods), as in **:case** or **:inverse-list** method combination. See the section "Examples Of **define-method-combination**", page 437.

(**:order** *vars...*)
: Specifies the order in which the component methods will be called, for the benefit of flavor examining tools. Each of the *vars* must be a variable bound by one of the *method-patterns*. Normally all of the *vars* bound by *method-patterns* would be included in the **:order** option, but this is not required. If no order is specified, the default is to use the *vars* in the order in which the *method-patterns* are written.

(**:method-transformer** *clauses...*)
: This can make changes to the arguments and body of methods for generic functions that use this type of method combination, and can control the validation of the arguments in the **defmethod** against the arguments in the **defgeneric**. Code in *clauses* receives arguments bound to the

following variables, and also has the parameters of the **define-method-combination** available to it:

sys:function-spec
The name of the method

flavor:method-combination
The method-combination type

flavor:method-arglist
The arglist specified for the method

flavor:method-body
The body of the method, including declarations

flavor:generic-method-arglist
The arglist specified for methods in the **defgeneric.**

Each *clause* consists of a keyword and a form evaluated to produce a replacement for a normal item. The clauses are:

:method-arglist
Replaces the argument list specified in **defmethod.**

:method-body
Replaces the body specified in **defmethod.**

:generic-method-arglist
Replaces the argument list for methods specified in **defgeneric.**

:inhibit-checking
If non-**nil**, inhibits comparing **flavor:method-arglist** and **flavor:generic-method-arglist** for the wrong number of arguments.

17.6.8 Interface to the Method Combination System

The interface takes care of calling the functions defined by the expansion of the **define-method-combination** special form, optimizing the results, and adding wrappers and whoppers to it to make the complete combined method.

The external interface to the method combination system consists of the following functions:

See the function **flavor:compose-handler** in *Symbolics Common Lisp: Language Dictionary*. See the function **flavor:compose-handler-source** in *Symbolics Common Lisp: Language Dictionary.*

17.7 Defining Functions Internal to Flavors

Flavors allows you to define functions that are lexically "inside" the flavor. Such functions have access to the instance variables of some instance of that flavor, or of a flavor built on it, defined by the value of **self**. There are two kinds of functions inside a flavor: methods and internal functions.

Methods	*Internal Functions*
Can be called from anywhere, via the generic function mechanism.	Can be called only from another function already inside the flavor. The caller can be a method or an internal function.
Can serve as entry-points or interfaces.	Can be used to share subroutines among methods.
Receive **self** as an argument.	Share the **self** of their caller.
Are defined with **defmethod**.	Are defined with **defun-in-flavor**, **defsubst-in-flavor**, and **defmacro-in-flavor**.
Have lexical access to instance variables of the object **self**.	Have lexical access to instance variables of the object **self**.
Are inherited from component flavors, using method combination rules.	Are inherited from component flavors, somewhat differently.

You can shadow a globally-defined function with an internal function with the same name. The globally-defined function can be an ordinary function or a generic function. However, common practice is to use unique names for internal functions, to minimize possible confusion.

17.7.0.1 Scoping of Internal Functions

The scoping of internal functions is as if each function inside the flavor were surrounded by a **flet** form that declared all of the internal functions. This is analogous to the scoping of instance variables. Internal functions are inherited from component flavors. If there are several internal functions with the same name, the first one in the ordering of flavor components is chosen.

If the name of an internal function of a flavor is used with the **function** special form (or the #' syntax), a closure is created that captures the value of **self** and the instance variables. This closure can be passed as a functional argument. Note that you use the name of the internal function, not its function spec.

Because internal functions of flavors are lexically scoped, they must be declared or defined before their callers. Otherwise the caller would not be compiled or evaluated in the proper lexical environment, and would not know that the name of the internal function refers to an internal function rather than to an ordinary function in the global environment. The **:functions** option to **defflavor** can be used to declare the names of **defun-in-flavor** internal functions of the flavor.

Currently the scope of internal functions does not include the initialization forms for **&optional**, **&key**, and **&aux** variables in **defmethod** and **defun-in-flavor** argument lists. This is a known bug.

17.7.0.2 Inheritance of Internal Functions

The inheritance of internal functions works differently than the inheritance of methods. The following example suggests that it is good practice not to give two internal functions for related flavors the same name:

```
(defflavor flav1 () ())
(defflavor flav2 () (flav1))
(defmethod (meth flav1) () 1)
(defmethod (meth flav2) () 2)
(defun-in-flavor (func flav1) () 1)
(defun-in-flavor (func flav2) () 2)
(defmethod (test flav1) () (list (meth self) (func)))

(test (make-instance 'flav1))
--> (1 1)

(test (make-instance 'flav2))
--> (2 1)
```

One might expect the last form to evaluate to (2 2), but it does not. In the method for **test**, the inheritance of **meth** depends on the actual flavor of **self** at run-time, but the inheritance of **func** depends only on the flavor for which the method that calls **func** is being defined, **flav1** in this case.

17.7.0.3 Redefining Internal Functions

You can redefine internal flavor functions by evaluating another **defun-in-flavor**, **defsubst-in-flavor**, or **defmacro-in-flavor** form. The new definition replaces the old.

Note that if you have defined an internal function, and decide to change it to a method, you must remove the definition of the internal function using **fundefine** and the function spec of the internal function.

17.7.0.4 Function Specs for Internal Functions

The function specs for internal functions, macros, and substitutable functions are described elsewhere: See the section "Function Specs for Flavor Functions", page 467.

Related Functions:

defun-in-flavor	Defines a function internal to a flavor.
defmacro-in-flavor	Defines a macro internal to a flavor.
defsubst-in-flavor	Defines a substitutable function internal to a flavor.

17.8 Wrappers and Whoppers

Wrappers and whoppers are used in certain cases in which **:before** and **:after** daemons are not powerful enough. **:before** and **:after** daemons let you put some code before or after the execution of a method; wrappers and whoppers let you put some code *around* the execution of the method. For example, you might want to bind a special variable to some value during the execution of a method. Or you might want to establish a condition handler, or set up a **catch** or **unwind-protect**. Wrappers and whoppers can also decide whether or not the primary method should be executed.

Whoppers are used more frequently than wrappers.

Wrappers	*Whoppers*
Similar to a macro.	Similar to a function.
If a wrapper is modified, all combined methods using it must be recompiled (this is done automatically).	If a whopper is modified, only the whopper must be recompiled.
The body of a wrapper is expanded in all the combined methods in which it is involved. The code is duplicated, not shared.	The body of a whopper is not expanded in multiple places.

Wrappers are slightly faster than whoppers.	Whoppers require two extra function calls each time they are called.

Because they involve the interaction of several complex mechanisms, you should use great care when using wrappers and whoppers.

The function specs for wrappers and whoppers are described elsewhere: See the section "Function Specs for Flavor Functions", page 467.

Changing and removing the definition of wrappers and whoppers is described elsewhere: See the section "Redefining Flavors, Methods, and Generic Functions", page 374.

17.8.1 Summary of Functions Related to Wrappers and Whoppers

defwrapper	Defines a wrapper.
defwhopper	Defines a whopper.
continue-whopper	Calls the methods for the generic function that was intercepted by the whopper. This is intended for use in **defwhopper** forms.
lexpr-continue-whopper	Like **continue-whopper**, but the last argument should be a list of arguments to be passed. This is useful when the arguments to the intercepted generic function include an **&rest** argument.
defwhopper-subst	Defines a wrapper by combining the convenient syntax of **defwhopper** with the efficiency of **defwrapper**.

17.8.2 Examples of Wrappers

The use of **defwrapper** is best explained by example. Suppose you need a lock locked during the processing of the **drain** generic function on an instance of the **cistern** flavor. The **drain** function takes one argument, *valve-position.* You have written a **lock-faucet** special form that knows how to lock the lock. **lock-faucet** needs to know the valve-position, the first argument to the **drain** function.

```
(defwrapper (drain cistern) ((valve-position) form)
  ;;; set lock for duration of method
  `(lock-faucet (self valve-position)
     ,form))                               ; Execute method itself
```

Note that the argument variable **valve-position** is not referenced with a comma preceding it. Argument variables are not bound at the time the

defwrapper-defined macro is expanded and the back-quoting is done. Rather, the result of that macroexpansion and back-quoting is a form that is evaluated with those variables bound to the arguments of the generic function, at the time the generic function is called.

Consider another example. You might want to run some code before executing the primary method. In addition, if the argument is **nil** you wanted to return from the generic function immediately, without executing the primary method. You could not do this using a **:before** daemon because **:before** methods are constrained to proceed to the primary method. You can use a wrapper to solve the problem. The following wrapper checks the argument and prevents anything further from happening, if it is **nil**. Instead of having a **:before** daemon, the "before" code is incorporated into the wrapper itself:

```
(defwrapper (drain cistern) ((valve-position) form)
  `(unless (null valve-position)  ; Do nothing if valve closed
     before-code                  ; Execute some "before" code
     ,form))                      ; Execute the body of the method
```

Suppose you need a variable for communication among the daemons for a particular generic function; perhaps the **:after** daemons need to know what the primary method did, and it is something that cannot be easily deduced from the arguments alone. You might use an instance variable for this, or you might create a special variable that is bound during the processing of the generic function, and used free by the methods. For example:

```
(defvar *communication*)
(defwrapper (drain cistern) (ignore form)
  `(let ((*communication* nil))
     ,form))
```

Similarly you might want a wrapper that puts a **catch** around the processing of a generic function so that any one of the methods could throw out in the event of an unexpected condition.

17.8.3 Examples of Whoppers

The following whopper adds code around the execution of the method that performs the generic function **print-integer** on instances of the **foo** flavor. Specifically, the whopper binds the value of the special variable **base** to 3 around the execution of the method. This function takes one argument, *n*:

```
(defwhopper (print-integer foo) (n)
  (let ((base 3))
    (continue-whopper n)))
```

The following whopper sets up a **catch** around the execution of the **compute-height** method of flavor **giant**, regardless of what arguments this methods accepts:

```
(defwhopper (compute-height giant) (&rest args)
  (catch 'too-high
    (lexpr-continue-whopper args)))
```

17.8.4 Mixing Flavors That Use Whoppers and Daemons

Like daemon methods, whoppers work in outward-in order; when you add a **defwhopper** to a flavor built on other flavors, the new whopper is placed outside any whoppers of the component flavors. However, *all* whoppers happen before *any* daemons happen. Thus, if a component defines a whopper, methods added by new flavors are considered part of the continuation of that whopper and are called only when the whopper calls its continuation.

17.8.5 Mixing Flavors That Use Wrappers and Whoppers

Whoppers and wrappers are considered equal for purposes of combination. If two flavors are combined, one having a wrapper and the other having a whopper for some method, then the wrapper or whopper of the flavor that is further out is on the outside. If, for some reason, the very same flavor has both a wrapper and a whopper for the same message, the wrapper goes outside the whopper.

17.8.6 Mixing Flavors That Use Wrappers and Daemons

When mixing flavors, wrappers work in outside-in order, just as daemons work. When you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. 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 wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

17.9 Complete Options For defflavor

This section describes the options to **defflavor**. For information on the format of the options: See the special form **defflavor**, page 354.

See the section "Inheritance Of **defflavor** Options", page 379.

:abstract-flavor Declares that the flavor exists only to define a protocol; it is not intended to be instantiated by itself. Instead, it is intended to have more specialized flavors mixed in before being instantiated.

Trying to instantiate an abstract flavor signals an error.

:abstract-flavor is an advanced feature that affects paging. It decreases paging and usage of virtual memory by allowing abstract flavors to have combined methods. Normally, only instantiated flavors get combined methods, which are small Lisp functions that are automatically built and compiled by the flavor system to call all of the methods that are being combined to make the effective method. Sometimes many different instantiated flavors use the same combination of methods. If this is the case, and the abstract flavor's combined methods are the same ones that are needed by the instantiated flavors, then all instantiated flavors can simply share the combined methods of the abstract flavor instead of having to each make their own. This sharing improves performance because it reduces the working set.

compile-flavor-methods is permitted on an abstract flavor. It is useful for combined methods that most specializations of that flavor would be able to share.

:area-keyword Changes the **:area** keyword to **make-instance** of this flavor to the supplied argument. This is useful if the flavor is using the **:area** keyword for some other purpose, such as an init keyword for an object's geometric or geographic area. This option is similar to the **:conc-name** option. Whereas **:conc-name** enables you to supply a prefix for an automatically generated function, the **:area-keyword** option to **defflavor** enables you to rename the **:area** keyword that is given to **make-instance** when making an instance of this flavor.

In rare cases, you might want to use the keyword **:area** for other purposes. For example, a geometric program might have a **triangle** flavor that has an instance variable named **area** that is equal to one-half of the product of its width and height. It is then important to distinguish between the instance variable **area** and the keyword that denotes in which area to create instances. You can rename that keyword for the affected flavor by providing the **:area-keyword** keyword to **defflavor**. For example:

```
(defflavor triangle (area)    ;instance var.
           ()
  :initable-instance-variables
  (:area-keyword :cons-area))
```

Now, when you make an instance, **:area** refers to the instance variable, and **:cons-area** indicates the area in which instances are to be created:

```
(make-instance 'triangle
               :area 44.23
               :cons-area GEOMETRY-AREA)
```

:component-order Enables you to state explicitly the ordering constraints for the flavor components whose order is important. You can use it to relax ordering constraints on component flavors for which order is not important. You can also use it to add ordering constraints on flavors that are not components; this means that if this flavor is later mixed with another flavor, the ordering of components takes into account the constraints given by this option.

If **:component-order** is given, the order of flavor components at the top of the **defflavor** form is no longer significant. The arguments to **:component-order** are lists. The members of each list are constrained to appear in the order they appear in the list. Any component that does not appear in these lists has no ordering constraints placed on it.

For example, the following form imposes many constraints on the ordering of the seven flavor components:

```
(defflavor foo (var1 var2)
           (a b c d e f g))
```

However, your program might not depend on a specific ordering of components, because the components have no effect on each other. For example, your program might depend only these ordering constraints:

- Flavor **c** must precede **d**.
- Flavor **b** must precede **g**.
- If flavor **x** is present (it could be a component of one of the components, or it could be mixed into flavor **foo** to create a new flavor), it must follow **b** and precede **g**.

You can specify those restrictions by giving the following option to **defflavor**:

```
(defflavor foo (var1 var2)
           (a b c d e f g)
  (:component-order (c d) (b x g)))
```

Note that this does not constrain flavors **c** and **d** to precede flavors **b**, **x**, and **g**. Also, it is not an error to specify an ordering constraint for a flavor that is not a component of this flavor. For example, it is valid to constrain the order of **x**, although **x** might not be a direct or indirect component of this flavor.

The way the flavor system determines the order of flavor components is described in detail elsewhere: See the section "Ordering Flavor Components", page 382.

:conc-name

Enables you to specify a prefix for the accessor functions created when the **:readable-instance-variables**, **:writable-instance-variables**, or **:locatable-instance-variables** options are supplied. Normally the accessor function to access the instance variable *v* of flavor *f* is named *f-v*, such as:

```
ship-mass
```

You can specify a different prefix as follows:

```
(defflavor ship (captain mass) ()
  (:conc-name get-)
  :readable-instance-variables)
```

The accessor functions for **ship** are thus named **get-captain** and **get-mass**.

You can use **:conc-name** to specify no prefix as follows:

```
(defflavor ship (captain mass) ()
  (:conc-name nil)
  :readable-instance-variables)
```

The accessor functions for **ship** are named **captain** and **mass**.

defflavor offers an alternate syntax enabling you to explicitly specify the names of accessor functions: See the section "Specifying Names for Functions That Access Instance Variables", page 464.

:constructor

Automatically generates a constructor function that enables you to create new instances of this flavor. The main advantage to constructor functions is that they are much faster than using **make-instance**. Whereas **make-instance** takes a flavor name argument and looks up information about the flavor, constructor functions have this information compiled into Lisp code.

The constructor function can take positional arguments instead of keyword arguments, or a mix of positional and keyword arguments. You can give the **:constructor** option more than once within a single **defflavor** to define several different constructors, each with its own arguments.

It is necessary to do a **compile-flavor-methods** to ensure that the constructor function is defined. The constructor function is defined as part of the macro-expansion of **compile-flavor-methods**, not part of the macro-expansion of **defflavor** (which makes it possible to define a flavor before all component flavors are defined). If you are incrementally developing code, you can put a **compile-flavor-methods** form into an editor buffer and use `c-sh-C` before running code that calls the constructor function.

The **:constructor** option takes two arguments. The first argument specifies the name of the constructor function. The second argument is used to describe what the arguments to the constructor will be. For example, a simple case like **(:constructor make-foo (a b c))** defines **make-foo** to be a three-argument constructor function whose arguments are used to initialize the instance variables named **a**, **b**, and **c**.

In addition, you can use the keywords **&optional**, **&rest**, **&key**, and **&aux** in the argument list. They work as you might expect, but note the following:

```
(defflavor foo (a b c d e f g h i) ()
  (:constructor make-foo
     (a &optional b (c 'sea)
        &rest d
        &key i
        &aux e (f 'eff))))
```

This defines **make-foo** to be a constructor that accepts one or more arguments. The first argument is used to initialize the **a** instance variable. The second argument is used to initialize the **b** instance variable. If there is no second argument, then the default value given in the body of the **defflavor** (if given) is used instead. The third argument is used to initialize the **c** instance variable. If there is no third argument, then the symbol **sea** is used instead. Any arguments following the third argument are collected into a list and used to initialize the **d** instance variable. If the keyword **:i** is supplied with a value, that value is used to initialize the **i** instance variable. If there are three or fewer arguments, then **nil** is placed in the **d** instance variable; the initial value of **e** is undefined; and the **f** instance variable is initialized to contain the symbol **eff**.

The actions taken in the **b** and **e** cases were carefully chosen to allow you to specify all possible behaviors. Note that the **&aux**

"variables" can be used to completely override the default initializations given in the **defflavor** form.

:default-handler The argument is the name of a function that is to be called when a generic function is called for which there is no method. The function is called with the arguments the instance was called with, including the message name; the values it returns are returned. If this option is not specified on any component flavor, it defaults to a function that signals an error.

The function specified with the **:default-handler** option to **defflavor** receives two additional arguments. The first argument is **self** and the second is always **nil**.

The following example shows the use of **:default-handler**.

```
(defflavor lisp-stream (forward) ()
  (:default-handler lisp-stream-forward))

(defun lisp-stream-forward (self ignore message &rest arguments)
  (lexpr-funcall (send self :forward) message arguments))
```

This is equivalent to defining a method for the **:unclaimed-message** message. See the message **:unclaimed-message** in *Symbolics Common Lisp: Language Dictionary*.

:default-init-plist Provides a list of alternating keywords and default value forms for keyword arguments that are allowed for **make-instance** of this flavor. The keywords can be: initable instance variables, init keywords, required init keywords, **:allow-other-keys**, or **:area**. If you do not specify one or more of these keywords as arguments to **make-instance**, they are set to the default in **:default-init-plist**. For the meaning of the **:allow-other-keys** and **:area** keywords: See the generic function **make-instance**, page 363.

This option allows one component flavor to default an option to another component flavor. The value forms are evaluated only when and if they are used. For example, the following would provide a default "frob array" for any instance for which the user did not provide one explicitly as an argument to **make-instance**:

```
(:default-init-plist :frob-array
                     (make-array 100))
```

Elements in the **:default-init-plist** that are init keywords are supplied to any methods defined for **make-instance** of this flavor. However, **:default-init-plist** elements that initialize instance variables are not supplied to **make-instance** methods.

:documentation The list of arguments to this option is remembered on the flavor's property list as the **:documentation** property. The convention for this list is as follows: A string describes what the flavor is for; this could consist of a brief overview in the first line followed by several paragraphs of detailed documentation. A symbol is one of the following keywords:

:mixin	A flavor that you might want to mix with others to provide a useful feature.
:essential-mixin	A flavor that must be mixed in to all flavors of its class, or inappropriate behavior follows.
:lowlevel-mixin	A mixin used only to build other mixins.
:combination	A combination of flavors for a specific purpose.
:special-purpose	A flavor used for some internal purpose by a particular program, which is not intended for general use.

:functions Declares the names of **defun-in-flavor** functions internal to the flavor. The arguments are names of internal functions.

defun-in-flavor functions are lexically scoped. They must either be defined before their callers, or declared by using the **:functions** option for **defflavor**. Otherwise the caller would not be compiled or evaluated in the proper lexical environment, and would not know that the name of the internal function refers to a **defun-in-flavor** function rather than to an ordinary function in the global environment. See the section "Defining Functions Internal to Flavors", page 443.

The **:functions** option is not appropriate for internal functions defined by **defsubst-in-flavor** or **defmacro-in-flavor**.

:gettable-instance-variables

This option is available for compatibility with the old message-passing flavor system. When writing new code, it is good practice to use the **:readable-instance-variables** option instead.

Enables automatic generation of methods for getting the values of instance variables. The message name is the name of the variable, in the keyword package (that is, put a colon in front of it.)

To use the automatically-generated method, use the old **send** syntax. For example:

```
(send my-ship :mass)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

:init-keywords Declares its arguments as keywords that should be accepted by **make-instance** of this flavor. Init keywords are:

- Keyword arguments to be processed by methods defined for **make-instance** for this flavor

  When you write a method for **make-instance** that accepts keyword arguments, you should include those keywords as init keywords. For related information: See the section "Writing Methods For **make-instance**", page 365.

- Keywords defined by the **:mixture** option to **defflavor**

  When you define a family of flavors, you should include any **:mixture** keywords as init keywords.

:init-keywords is used for error-checking when you create an instance. All keywords given to **make-instance** are either initable instance variables or init keywords, or keywords accepted by **make-instance**, like **:area**. If the caller misspells a keyword or otherwise uses a keyword that no component flavor handles, **make-instance** signals an error.

Note that whenever you have a **:required-init-keywords** clause containing keywords that are to be used by **make-instance** methods, it is necessary to include those keywords in the **:init-keywords** clause as well.

It is legal but pointless to include keywords that initialize instance variables as init keywords.

:initable-instance-variables
Enables you to initialize the specified instance variables when

making an instance of this flavor. The instance variables are sometimes called *initable*. In the **make-instance** form, you initialize values by including the keyword (name of the instance variable) followed by its value, as follows:

```
(make-instance flavor-name :variable-name value)
```

For example, when making an instance of the **ship** flavor, we can initialize the **mass** instance variable (assuming it is initable), as follows:

```
(make-instance 'ship :mass 105)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

If you want to specify that an instance variable inherited from another component flavor should be initable, you can include the name of that instance variable explicitly in the top of the **defflavor** form. This option checks for spelling errors, so any instance variables that are declared initable must appear at the top of the **defflavor** form, or be in the list of **:required-instance-variables**.

defflavor offers an alternate syntax enabling you to explicitly specify the keyword to be used to initialize an instance variable: See the section "Specifying Names for Functions That Access Instance Variables", page 464.

:locatable-instance-variables

Enables you to use **locf** to get a locative pointer to the cell inside an instance that contains the value of an instance variable as follows:

```
(locf (accessor object))
```

If the accessor function is **ship-mass**, and the object is **my-ship**, you can get a locative pointer to the cell inside **my-ship** containing the value of the **mass** instance variable as follows:

```
(locf (ship-mass my-ship))
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

Any instance variables specified by **:locatable-instance-variables** are automatically made readable as well; that is, an accessor function is automatically defined.

defflavor offers an alternate syntax enabling you to explicitly specify the names of accessor functions: See the section "Specifying Names for Functions That Access Instance Variables", page 464.

:method-combination

Declares the way that methods from different flavors are to be combined. The syntax is:

The syntax is:

```
(:method-combination
     generic-function name
     generic-function (name args...)
     ...)
```

If the **:method-combination** option is also supplied to **defgeneric**, that option must agree with the **:method-combination** option given to **defflavor**.

For further details on usage and an example: See the section "Using The **:method-combination** Option", page 420.

:method-order The arguments are names of generic functions that are frequently used or for which speed is important. Their handlers are inserted into the handler hash table first, so that they are found by the first hash probe.

:mixture Defines a family of related flavors. The organization of this family usually consists of one *basic flavor* and several *mixin flavors*. The flavors in the family are automatically constructed by mixing various mixins with the basic flavor. When **make-instance** is called, it uses its keyword arguments (or defaults to values in the **:default-init-plist** of the **defflavor** form) to choose which flavor of the family to instantiate.

The basic flavor is the one that includes the **:mixture** option in its **defflavor**. By convention, it is often named **basic-***foo*. Often the basic flavor is not intended to be instantiated. If so, you should supply the **:abstract-flavor** option to specify that.

The names for the family members are chosen automatically. The name of such an automatically constructed flavor is a concatenation of the names of its components, separated by hyphens. Note that obvious redundancies are removed heuristically. An example is shown later in this section.

defflavor of the basic flavor defines the automatically constructed flavors as well as the basic flavor. Similarly, **compile-flavor-methods** of the basic flavor also compiles combined methods of the automatically constructed flavors.

The **:mixture** option is not inherited by flavors from their component flavors. You can still build a new flavor from a flavor that uses **:mixture**, but you should not expect that the new flavor follows the conventions specified by the **:mixture** option of its component. If you try to apply the **:mixture** conventions of a component flavor to the new flavor built on it, you will get a warning. If you want the new flavor to follow the same conventions, you can include the same **:mixture** option in the **defflavor** of the new flavor.

The **:mixture** option has the following form:

(:mixture *spec spec* ...)

Each *spec* is processed independently, and all the resulting mixins are mixed together. A *spec* can be any of the following:

(*keyword mixin*)
: Add mixin if the value of *keyword* is **t**; add nothing if **nil**.

(*keyword* (*value mixin*) (*value mixin*) ...)
: Look up the value of *keyword* in this alist and add the specified mixin.

(*keyword mixin subspec subspec* ...)

(*keyword* (*value mixin subspec subspec* ...) ...)

A *subspec* has the same form as a spec. Subspecs are processed only when the specified keyword has the specified value. Use them when there are interdependencies among keywords.

A *mixin* is one of the following:

symbol	The name of a flavor to be mixed in.
nil	No flavor needs to be mixed in if the keyword takes on this value.
string	This value is invalid: Signal an error with the string as part of the message.

A *value* can be anything that is acceptable as a clause key for

case. This includes symbols, numbers, characters, instances, and named structures; but excludes lists, strings, and arrays other than named structures. The symbol **otherwise** is treated specially, as in **case**. For example, if you want to allow for values of **:size** other than 1, the **:mixture** clause is:

```
(:mixture (:size (1 small-mixin)
                 (otherwise nil)))
```

make-instance checks that the keywords are given with valid values, if you do not have **otherwise** as a clause key.

You need an **:init-keywords** declaration for any keywords that are used only in the **:mixture** declaration.

The following example defines a basic flavor called **cereal-stream**:

```
(defflavor cereal-stream (...) (stream)
  ...
  :abstract-flavor
  (:init-keywords :characters :direction
                  :ascii :hang-up-when-close)
  (:mixture (:characters
             (t nil (:direction
                      (:in buffered-line-input-stream)
                      (:out buffered-output-character-stream))
                    (:ascii ascii-translating-character-stream))
             (nil nil (:direction (:in buffered-input-stream)
                                  (:out buffered-output-stream))
                      (:ascii "Ascii translation is not
                               meaningful for binary streams")))
           (:hang-up-when-close hang-up-when-close-mixin)))
```

The declaration above indicates that the basic flavor **cereal-stream** cannot be instantiated alone. The **:direction** option and an appropriate value (**:in** or **:out**) must be provided to **make-instance**. The **:characters** option does not itself add any mixins (hence the **nil**), but the processing of the **:direction** option depends on the value of the **:characters** option, which selects a character stream or a binary stream. The **:ascii** option is allowed only for character streams, and an error message is specified if it is used with a binary stream. If **:ascii** had not been mentioned in the **:characters nil** case, the keyword would have been ignored by **make-instance** on the assumption that a **make-instance** method was going to use it. Any kind of **cereal-stream** can have a **:hang-up-when-close** option.

You could make an instance of a member of this family as follows:

```
(make-instance 'cereal-stream :characters t
                              :direction :in
                              :hang-up-when-close t)
```

The name of the flavor that is instantiated is: **hang-up-when-close-buffered-line-input-cereal-stream**.

:no-vanilla-flavor Normally when a flavor is defined, the special flavor **flavor:vanilla** is included automatically at the end of its list of components. The **flavor:vanilla** flavor provides some default methods for several useful functions that all objects are supposed to understand, including **sys:print-self** and **:which-operations**, among others.

If any component of a flavor specifies the **:no-vanilla-flavor** option, **flavor:vanilla** is not included in that flavor. This option should not be used casually.

:ordered-instance-variables

This option is intended for internal system uses only. **:ordered-instance-variables** increases efficiency at the cost of dynamic modification. The arguments are names of instance variables that must appear first (and in this order) in all instances of this flavor or any flavor depending on this flavor. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

If you try to redefine a flavor that has ordered instance variables, you will notice that the actual order of the instance variables does not change once the Flavors system has committed to a particular order. Flavors gives you a warning when this occurs. However, you can change the order by reloading the code into a world in which the flavor has not yet been defined.

:readable-instance-variables

Creates an accessor function for querying an object for the value of each of the specified instance variables. The name of the function is the name of the flavor, followed by a dash "-", followed by the name of the instance variable. The accessor function for the instance variable *v* of flavor *f* is:

f-v

For example, the **ship** flavor has an instance variable named **mass**. If the **:readable-instance-variables** option is given, an accessor function named **ship-mass** is created. You can use it on an instance of **ship** as follows:

```
(ship-mass my-ship)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

The accessor function is created in the current package.

You can use the **:conc-name** option to **defflavor** to specify a different name for the accessor function.

defflavor offers an alternate syntax enabling you to explicitly specify the names of accessor functions: See the section "Specifying Names for Functions That Access Instance Variables", page 464.

:required-flavors Specifies flavors that must be included as components (directly or indirectly) in a new flavor incorporating this one, if the flavor is to be instantiated. The arguments are names of the required flavors. Typically the **:required-flavors** option is used when defining a mixin flavor to specify which base flavor or flavors it requires.

The difference between giving the **:required-flavors** option and listing them directly as components at the top of the **defflavor** form is that the **:required-flavors** option does not make any commitments about where those flavors should appear in the ordered list of components. The order of the **:required-flavors** is determined by the flavor that does specify them as components.

Declaring a flavor as required using the **:required-flavors** option

- Allows instance variables declared by that flavor or its components to be accessed.

- Allows you to use any internal functions defined for the flavors in the **:required-flavors** clause, or for their components. See the section "Defining Functions Internal to Flavors", page 443.

- Provides error checking. An attempt to instantiate a flavor that does not include the required flavors as components signals an error.

:required-init-keywords

Specifies keywords that must be supplied when making an instance of this flavor. The arguments are the required keywords. It is an error to try to make an instance of this flavor or incorporate it without specifying these keywords as arguments to **make-instance** or as a **:default-init-plist** option in a component flavor. This error is often detected at compile-time.

Note that whenever you have a **:required-init-keywords** clause containing keywords that are to be used by **make-instance** methods, it is necessary to include those keywords in the **:init-keywords** clause as well.

:required-instance-variables

Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. The arguments are the required instance variables. It is an error to try to make an instance of this flavor or a flavor incorporating it if it does not have the required instance variables.

Required instance variables can be freely accessed by methods just like normal instance variables. The difference between using the **:required-instance-variables** option and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-methods

Specifies generic functions that must be supported with methods by a new flavor incorporating this one, if the flavor is to be instantiated. The arguments are the names of the generic functions that must be supported with methods.

It is an error to instantiate such a flavor if it lacks a method for one of these generic functions. When the **:required-methods** option is given, it is possible for the error to be detected when the flavor is defined (usually at compile-time), rather than at run-time.

Typically this option appears in the **defflavor** form for a base flavor. Usually this is used when a base flavor calls a generic function on itself, but the base flavor does not provide a method for that generic function. **:required-methods** indicates that the base flavor cannot be instantiated alone, but must be instantiated with other components (mixins) that do handle the required generic functions.

:settable-instance-variables

This option is available for compatibility with the old message-passing flavor system. When writing new code, it is good practice to use the **:writable-instance-variables** option instead.

Enables automatic generation of methods for setting the values of instance variables. The message name is "**:set-**" followed by the name of the variable. All settable instance variables are also automatically made gettable and initable.

To use the automatically-generated method, use the old **send** syntax. For example:

```
(send my-ship :set-mass 100)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

:special-instance-variables

This option is intended for internal system uses only. It is documented for completeness.

Use the **:special-instance-variables** option if you need instance variables to be bound as special variables to values in the instance, when certain methods are called. (The methods are specified with **:special-instance-variable-binding-methods.**) This option detracts from performance and should be avoided.

The format of **:special-instance-variables** is the same as that of **:readable-instance-variables**. If the option is given alone, all instance variables are bound. If the option is given in the format (**:special-instance-variables a b c**), only the variables **a**, **b**, and **c** are bound.

:special-instance-variable-binding-methods

This option is intended for internal system uses only. It is documented for completeness.

This option specifies names of generic functions and messages that should cause any instance variables declared **:special-instance-variables** to be bound as special variables to values in the instance.

:writable-instance-variables

Enables you to use **setf** to set the value of an instance variable as follows:

```
(setf (accessor object) value)
```

If the accessor function is **ship-mass**, and the object is **my-ship**, you can set the value of the **mass** instance variable to 100 as follows:

```
(setf (ship-mass my-ship) 100)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

Any instance variables specified by **:writable-instance-variables** are automatically made readable as well. See the description of the **:readable-instance-variables** option to **defflavor**.

defflavor offers an alternate syntax enabling you to explicitly specify the names of accessor functions: See the section "Specifying Names for Functions That Access Instance Variables", page 464.

17.9.1 Specifying Names for Functions That Access Instance Variables

This section describes an alternate format of **defflavor** that enables you to explicitly specify the name of the generic functions that read, write, or locate an instance variable. Using this format, you specify the name of the reader function; note that the name of the writer function always involves **setf** and the locator function involves **locf**. You can also explicitly specify the keyword used to initialize an instance variable. If you are using message-passing, this flexible syntax also applies to gettable and settable instance variables.

This example exhibits the syntax of all of these:

```
(defflavor test-flavor (a b c d e f g h i j k l m n o p) ()
  (:initable-instance-variables a b (:sea c))
  (:settable-instance-variables d e (:change-f f))
  (:gettable-instance-variables g h (:eye i))
  (:readable-instance-variables j (get-kay k))
  (:writable-instance-variables l (m-value m))
  (:locatable-instance-variables n (get-o o)))
```

When making an instance of **test-flavor**, you can initialize the variables **a**, **b**, and **c** as follows:

```
(setq test-instance (make-instance 'test-flavor :a 2
                                                 :b 4
                                                 :sea 6))
```

You can set the value of the variables **d**, **e** and **f** as follows:

```
(send test-instance :set-d 22)
(send test-instance :set-e 44)
(send test-instance :change-f 66)
```

You can get the value of the variables **g**, **h** and **i** as follows:

```
(send test-instance :g)
(send test-instance :h)
(send test-instance :eye)
```

You can read the values of the variables **j**, **k**, **l**, **m**, **n** and **o** as follows:

```
(test-flavor-j test-instance)
(get-kay test-instance)
(test-flavor-l test-instance)
(m-value test-instance)
(test-flavor-n test-instance)
(get-o test-instance)
```

You can write the value of the variable **m** as follows:

```
(setf (m-value test-instance) 33)
```

You can locate the value of the variables **n** and **o** as follows:

```
(locf (test-flavor-n test-instance))
(locf (get-o test-instance))
```

17.9.2 Specifying defflavor Options Twice

Some **defflavor** options are allowed to be specified twice. Those that are not allowed to be specified twice cause a warning to occur if you do so. This is the list of options that can be specified twice:

- **:constructor**
- **:default-init-plist**
- **:functions**
- **:gettable-instance-variables**
- **:init-keywords**
- **:initable-instance-variables**
- **:locatable-instance-variables**
- **:method-combination**
- **:method-order**
- **:ordered-instance-variables**
- **:readable-instance-variables**
- **:required-flavors**
- **:required-init-keywords**
- **:required-instance-variables**
- **:required-methods**
- **:settable-instance-variables**
- **:special-instance-variables**
- **:special-instance-variable-binding-methods**
- **:writable-instance-variables**

17.10 Advanced Concepts For defmethod

17.10.1 defmethod Declarations

It is legal to give the same **declare** statements to **defmethod** as are accepted by **defun**. **defmethod** also accepts these additional **declare** statements:

(declare (flavor:solitary-method))

If this declaration is used, and only one method is available for the generic function, Flavors implements the generic function with the goal of saving space, and at the expense of making the method slower to call. The generic function calls the method directly, bypassing the usual dispatching mechanism.

If more than one method is available for the generic function, this declaration has no effect. Also, this declaration saves space only when a method is inherited by more than one flavor.

The input editor illustrates the use of solitary methods. Each input editor command is a method. There are many commands, and there is no need for them to be any faster than the user can type them. Declaring these commands solitary methods optimizes space.

(declare (flavor:inhibit-arglist-checking))

Normally, Flavors checks the arguments of **defmethod** forms to ensure that they are consistent with the arguments expected by the generic function. This declaration prevents that consistency check from occurring. For example:

```
(defmethod (accept encapsulating-output-stream)
           (presentation-type &rest args)
  (declare (flavor:inhibit-arglist-checking))
  (lexpr-send stream 'accept presentation-type :stream
              self args))
```

17.10.2 Implicit Blocks for Methods

The interpreter and compiler generate implicit blocks for functions whose name is a list (such as methods) just as they do for functions whose name is a symbol. You can use **return-from** for methods. The name of a method's implicit block is the name of the generic function it implements. If the name of the generic function is a list, the block name is the second symbol in that list.

17.10.3 Variant Syntax Of defmethod

The following variant **defmethod** syntax is supported, but rarely used:

(defmethod (*generic-function flavor options..*) *symbol*)

symbol is the name of an ordinary function (not a generic function) that is to be used as the method for performing *generic-function* on instances of the given *flavor.*

For example:

```
(defmethod (delete-file logical-pathname) logical-pathname-pass-on)
```

symbol cannot be an internal flavor function defined by **defun-in-flavor, defmacro-in-flavor**, or **defsubst-in-flavor.** You can use the body of the normal **defmethod** syntax to call an internal flavor function to perform the operation.

17.11 Function Specs for Flavor Functions

This section tells what the function specs are for various types of flavor functions, such as generic functions, methods, and wrappers. For more detailed information on how to use function specs: See the section "Function Specs", page 251.

Type of Function	*Function Spec*
Combined method	**(flavor:combined** *generic-function flavor***)**
Generic function	Its name, usually a symbol
Handler	**(:handler** *generic-function flavor***)**
Internal function	**(defun-in-flavor** *function-name flavor***)**
Internal macro	**(defun-in-flavor** *macro-name flavor***)**
Internal substitutable function	**(defun-in-flavor** *subst-name flavor***)**
Locator function	**(locf** *function***)**
Method	**(flavor:method** *generic-function flavor options..***)**
Setter function	**(setf** *generic-function***)**
Whopper	**(flavor:whopper** *generic-function flavor***)**
Wrapper	**(flavor:wrapper** *generic-function flavor***)**

A note to Zetalisp programmers: the function spec of a setter function is **(setf** *generic-function***)**, not **(zl:setf** *generic-function***)**.

17.11.1 Setter and Locator Function Specs

A setter function is a generic function that sets an instance variable. A locator function is a generic function that locates an instance variable.

You invoke a setter function by typing something like:

```
(setf (ship-mass my-ship) 100)
```

The function spec for the above setter function is **(setf ship-mass)**.

Similarly, you invoke a locator function by typing:

```
(locf (ship-mass my-ship))
```

The function spec for the above locator function is **(locf ship-mass)**.

A setter function can be defined in the following ways:

- Automatically: by providing the **:writable-instance-variables** option for **defflavor**. For example:

  ```
  (defflavor ship (x-velocity y-velocity mass)
             ()
    :writable-instance-variables)
  ```

- Implicitly: by using **defmethod** to define a method, which in turn creates a generic function. For example:

```
(defmethod ((setf ship-mass) ship) (new-mass)
  (setq mass new-mass
        mass-squared (* new-mass new-mass))
  new-mass)
```

- Explicitly: by defining the generic function **(setf ship-mass)** using **defgeneric**. This would be unusual.

It is not necessary to use **defsetf** to tell **setf** how to deal with setter functions such as **ship-mass**. The existence of the function named **(setf ship-mass)** is enough for **setf** to know what to do.

Note that not all setter functions have the function spec **(setf function)**. Only those that were defined in one of the three ways described above have such a function spec. Setter functions defined with **defsetf** do not have such a function spec. Here is an example of using the setter function spec:

```
(let ((setter (if cond #'(setf foo) #'(setf bar))))
  (dolist (thing things)
    (funcall setter thing nil)))
```

17.12 Property List Methods

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 section 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.

sys:property-list-mixin *Flavor*

This mixin flavor provides methods that perform the generic functions on property lists. **sys:property-list-mixin** provides methods for the following generic functions:

:get *indicator* *Message*

The **:get** message looks up the object's *indicator* property. If it finds such a property, it returns the value; otherwise it returns **nil.**

:getl *indicator-list* *Message*

The **:getl** message is like the **:get** message, except that the argument is a list of indicators. The **:getl** message searches down the property list for any of the indicators in *indicator-list* until it finds a property whose indicator is one of those elements. It returns the portion of the property list beginning with the first such property that it found. If it does not find any, it returns **nil.**

:putprop *property indicator* *Message*

Gives the object an *indicator*-property of *property.*

:remprop *indicator* *Message*

Removes the object's *indicator* property by splicing it out of the property list. It returns that portion of the list inside the object of which the former *indicator*-property was the **car.**

:push-property *value indicator* *Message*

The *indicator*-property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This message sets the *indicator*-property of the object to a list whose **car** is *value* and whose **cdr** is the former *indicator*-property of the list. This is analogous to doing:

```
(zl:push value (get object indicator))
```

See the macro **zl:push** in *Symbolics Common Lisp: Language Dictionary.*

:property-list *Message*

Returns the list of alternating indicators and values that implements the property list.

:set-property-list *list* *Message*

Sets the list of alternating indicators and values that implements the property list to *list.*

:property-list *list* (for **sys:property-list-mixin**) *Init Option*

Initializes the list of alternating indicators and values that implements the property list to *list.*

17.13 Generic Functions and Messages Supported By flavor:vanilla

This section lists the generic functions and messages for which **flavor:vanilla** provides a method.

Any flavor can override a **flavor:vanilla** method by providing a method for one of the operations listed below.

You can write a method for **flavor:vanilla.** No method defined for **flavor:vanilla** can access instance variables, because the flavor has none. Note that when you write a method for **flavor:vanilla,** it takes a long time for the Flavor system to update all flavors dependent on **flavor:vanilla.**

flavor:vanilla *Flavor*

This flavor is included in all flavors by default. **flavor:vanilla** has no instance variables, but it provides several basic useful methods, some of which are used by the Flavor tools.

Every flavor has **flavor:vanilla** as a component flavor, unless you specify not to include **flavor:vanilla** by providing the **:no-vanilla-flavor** option to **defflavor**. It is unusual to exclude **flavor:vanilla**.

flavor:vanilla provides methods for the following generic functions:

:describe	Calls **flavor:describe-instance**, which prints the following information onto the ***standard-output*** stream: a description of the instance, the name of its flavor, and the names and values of its instance variables. It returns the instance.
get-handler-for	Returns the given object's method for a specified operation, or **nil** if the object has no method for the operation.
:operation-handled-p	Returns **t** if the given object has a handler for the specified operation, **nil** if it does not.
sys:print-self	Produces a printed representation of the given object.
:send-if-handles	Performs the specified generic function if the given object has a method defined for it; otherwise returns **nil**.
:which-operations	Returns a list of generic functions and messages that the given object supports with methods.

17.14 Using Message-Passing Instead of Generic Functions

Message-passing is supported for compatibility with previous versions of the flavor system. This section describes the features that support message-passing. When writing new programs, it is good practice to use generic functions instead of message-passing.

17.14.1 Defining Methods to Be Called by Message-Passing

The syntax for **defmethod** is as follows:

```
(defmethod (generic-function flavor options..) (arg1 arg2..) body..)
```

To define a method to be invoked by sending a message (instead of calling a generic function), supply a keyword as the *generic-function* argument to **defmethod**. The keyword is the name of the message. For example:

```
;;; define a message :speed-of, to be used with send syntax
;;; on instances of the ship flavor

(defmethod (:speed-of ship) ()
  (sqrt (+ (expt x-velocity 2)
           (expt y-velocity 2))))
```

This method should be invoked by the old **send** syntax:

```
(send instance message args...)
```

For example:

```
(send my-ship :speed-of)
```

You can also specify that any methods for a certain flavor should be invocable both by generic functions and messages. To do so, supply the **:compatible-message** option to **defgeneric**. Thus, any methods for that generic function can be called with the generic function syntax, or the old **send** syntax.

For any methods invocable by messages, you can call **defgeneric** to update the flavor to treat those methods as generic functions. If you do so, the old **send** syntax no longer works.

17.14.2 Defining a Compatible Message for a Generic Function

The **:compatible-message** option to **defgeneric** indicates that any methods written for this generic function should be callable in two ways: by calling the generic function or by sending a message. The name of the generic function is given as an argument to **defgeneric**. The name of the message is given as an argument to the **:compatible-message** option.

For example:

```
(defgeneric speed (moving-objects)
   (:compatible-message :speed-of))

(defmethod (speed ship) ()
  (sqrt (+ (expt x-velocity 2)
           (expt y-velocity 2))))
```

You can invoke the generic function **speed** as follows:

```
(speed my-ship)
```

You can invoke the same method by sending the **:speed-of** message as follows:

```
(send my-ship :speed-of)
```

You can also use either **speed** or **:speed-of in defmethod; :speed-of** is automatically changed to **speed** in the method's function spec.

If you are converting message-passing programs to generic functions, and using **:compatible-message** to define messages so that old programs that use those messages will continue to work, you should be sure that every use of the message you are defining is within your program. If other programs are using the same message name for different purposes, your **defgeneric** form will have an effect on the other methods. For example, all methods for that message are constrained to take the arguments that your generic function takes. Also, when any methods for that message are compiled, they are defined as being methods for your generic function.

17.14.3 Functions for Passing Messages

send	Sends a message to a flavor instance.
lexpr-send	Like **send**, except that the last argument should be a list. All elements of that list are passed as arguments.
send-if-handles	Sends a message to a flavor instance, if the flavor has a method defined for this message.
lexpr-send-if-handles	Like **send-if-handles**, except that the last element of arguments should be a list. All elements of that list are passed as arguments.

18. Table Management

18.1 Introduction to the Table Management Facility

A *table* is a data structure that consists of some number of *entries*, each containing one or more objects. The number of objects per entry is fixed and uniform in any given table. The simplest tables consist of entries which are *keys*. In the most common table, the first object in each entry of a table is the key, and the second object is the *value*. In more complex tables, there can be some combination of multiple keys and multiple values.

This sample table is made up of key and value pairs, where the key is the bird type and the value is a list of foods that bird eats:

	KEY (bird)	VALUE (diet)
	blue-heron	(frogs snakes turtles)
ENTRY	horned-owl	(mice snakes)
	pelican	(fish)
	...	...

The principal operations on tables are:

- searching by key
- inserting and deleting entries
- examining all entries
- deleting all entries

Some tables also support the additional operations of retrieving the first entry, retrieving the last entry, and possibly retrieving the entries in order by key.

The table management facility performs these operations on tables of many forms, using one common interface. Thus, you need not worry about the internal representation of the data or other properties of the table. If you create tables with this facility, you have code which is easily ported to Common Lisp while taking advantage of the efficiencies provided by the facility. If you create tables which do not use the Symbolics extensions to the **make-hash-table** function, your code will already be compatible with Common Lisp.

There are features and advantages to using this scheme for both simple and complex tables.

For the user who will only be making simple tables, this facility has the advantage of being portable to any other Common Lisp implementation while the underlying structure is invisible.

For the more advanced uses of tables there are other advantages. It is easy to extend this facility to create tables with your own mixins for customization.

Tables created with the Common Lisp function, **make-hash-table**, have a number of performance benefits. They use the internal representation best suited to the data in the table and they have a number of optimizations built in that make them highly efficient.

Tables created with the Zetalisp function, **zl:make-hash-table**, still work in this release, but may not work in a future release. The table management facility is a replacement for Zetalisp hash tables, and provides the same functionality in a more efficient way. If you have code which uses these functions, you should consider converting them to the new facility.

The Lisp functions that operate on lists, arrays, and other structures that are used as tables remain the same. The traditional creation functions can still be used to make any tables that you do not want to make with this facility. Even though tables made from these structures can be constructed and managed via the table management facility, there are some times that old style tables can be more useful. These features are covered later. See the section "Other Data Types Used as Tables", page 484.

18.2 Table Management Interface

This section covers the table management operations and interface. The interface you use to create and access tables is the Common Lisp hash table interface. It has been extended to support more functionality, but the basic framework is the same.

It is called the "hash table interface" because it uses the Common Lisp hash table functions, such as **make-hash-table**, **gethash**, **hash-table-count**, and **sxhash**. Genera hash tables do not necessarily hash, they only use hashing when the table requires it.

18.2.1 Hash Table Interface to the Table Management Facility

For the most common types of tables, a table object is a Lisp object that is similar to an association list. Every table object has a set of *entries*, each of which associates a particular *key* with a particular *value*. The basic functions that operate on table objects create entries, delete entries, and find the value that is associated with a given key.

A given table object can only associate one value with a given key; if you try to add a second value it replaces the first.

18.2.1.1 Table Functions

clrhash *table*	Returns *table* after removing all of its entries.
si:equal-hash *x*	Computes the hash code for *x*, and returns it as an integer.
gethash *key table*	Returns three values; *value*, whether or not *key* was found and the found *key*.
hash-table-count *table*	Returns the number of entries currently in *table*.
hash-table-p *object*	Returns **t** if *object* is a table object.
make-hash-table	Creates a new table object.
maphash *function table*	For every entry in *table*, calls *function* on the key of the entry and the value of the entry.
modify-hash *table key function* &rest *args*	Finds the value associated with *key* in *table*, then calls *function* with *key*, this value, a flag indicating whether or not the value was found, and *args*; and puts whatever is returned by the call to *function* into *table*, associating it with *key*.
remhash *key table*	Tries to remove the entry associated with *key*, and returns **t** if the entry was removed or **nil** if none was found.
setf *place newvalue*	Used in combination with **gethash** to create a new entry in a table.
swaphash *key value hash-table*	Creates an entry in *hash-table* associating *key* to *value*. If there is an existing entry for *key*, then it replaces the value of that entry with *value*.
table-size *table*	Returns the total number of entries in *table*.

You create table objects with the **make-hash-table** function, which takes various initialization options. You can add new entries to table objects by using the **gethash** function with the **setf** macro. To look up a key and find the associated value, use the **gethash** function. An example of how to create and work with a table that describes a plant follows:

```
(setq plant (make-hash-table :size 10))
   => #<EQL-ALIST-PROCESS-LOCKING-DUMMY-GC-LOCKING-ASSOCIATION-
        MUTATING-TABLE 1320453>

(setf (gethash 'name plant) 'african-violet) => AFRICAN-VIOLET

(setf (gethash 'genus plant) 'saintpaulia) => SAINTPAULIA

(setf (gethash 'flowers plant) t) => T

(setf (gethash 'flower-colors plant) '(violet white pink))
   => (VIOLET WHITE PINK)

(hash-table-count plant) => 4

(describe plant)
   => #<EQL-ALIST-PROCESS-LOCKING-DUMMY-GC-LOCKING-ASSOCIATION-
        MUTATING-TABLE 1320453> is a table with 4 entries.
   Test function for comparing keys = EQL, hash function =
        CLI::XEQLHASH
   Do you want to see the contents of the hash table? (Y or N) Yes.
   Do you want it sorted? (Y or N) Yes.
      FLOWER-COLORS → (VIOLET WHITE PINK)
      FLOWERS → T
      GENUS → SAINTPAULIA
      NAME → AFRICAN-VIOLET
   #<EQL-ALIST-PROCESS-LOCKING-DUMMY-GC-LOCKING-ASSOCIATION-
     MUTATING-TABLE 1320453>
```

In this example we first create a table with a **:size** of **10**, then bind it to the symbol **plant**. The next four forms add information about the new plant to the new table. Each of these forms creates an association between two lisp objects. The function **hash-table-count** returns the number of entries in the table. The function **describe**, when given a table object returns some useful information about that object.

This table object has four entries in it: the first associates from the symbol **name** to the symbol **african-violet,** the second associates from the symbol **genus** to the symbol **saintpaulia**, the third associates from the symbol **flowers** to the symbol **t**, and the last associates from the symbol **flower-colors** to the list **(violet white**

pink). The symbols **name, genus, flowers** and **flower-colors** are keys, and the symbols **african-violet, saintpaulia,** and **t**, and the list **(violet white pink)** are their associated values. As you can see, keys do not have to be symbols; they can be any Lisp object. Likewise values can be any Lisp object.

```
    (gethash 'flower-colors plant)
      => (VIOLET WHITE PINK) and T and FLOWER-COLORS

    (gethash 'name plant)
      => AFRICAN-VIOLET and T and NAME

    (gethash 'leaves plant)
      => NIL and NIL and NIL
```

The three values returned by **gethash** are the value associated with the key, a boolean value for whether or not the key was found, and the key itself (if found). The third value (the key) is a Symbolics extension to Common Lisp.

When a table object is first created, it has a *size*, which is the number of entries it can hold. This number is an estimate used to set up the internal representation of the table. If the number of entries exceeds the current *size*, the table object automatically grows.

One of the new features of this facility is that tables change internal representation based on the initial keywords to **make-hash-table**, the current size of the table, and type of the data set. This means that the table changes at run-time as the table grows and shrinks. For example, if the internal representation of the table is an alist, and it grows past an efficient size for alists, the table management facility automatically changes it to a hash or block array table, unless otherwise specified in the call to **make-hash-table**.

Table object may be saved into files since they support the **:fasd-form** methods required to dump their data to a binary file using the function **sys:dump-forms-to-file**. See the function **sys:dump-forms-to-file** in *Reference Guide to Streams, Files, and I/O.*

If you need to access each entry of a table in succession, there are provisions for iterating over entries in tables with the **loop** iteration macro. See the section "**loop** Iteration Over Hash Tables or Heaps", page 555.

18.2.2 Creating Table Objects

You use **make-hash-table** to make a new table, as described earlier. See the section "Hash Table Interface to the Table Management Facility", page 476. Many initialization options to **make-hash-table** are available in order to customize a table to your application.

:test	Determines how keys are compared. Its argument should be a symbol; **eql** is the default. The tables are optimized for the symbols **eq**, **eql**, and **equal**. The function values **#'eq**, **#'eql**, and **#'equal** are accepted for Common Lisp compatibility, but are mapped to the equivalent symbol. You can also specify some arbitrary predicate for key comparison. For example, if you intend to use the table for numbers, then the predicate = might be more appropriate. The **:test** and **:size** keywords control the table's initial internal representation.
:size	Sets the initial size of the table. This keyword approximates the number of entries, since the number you specify can be rounded up to the next good size for the hashing algorithm. The **:test** and **:size** keywords control the initial internal representation.
:rehash-size	Sets the growth factor of the table when it becomes full. If the value is an integer, it specifies the number of entries to add. If it is a floating-point number, it specifies the ratio of the new size to the old size. If the value is neither an integer or a floating-point number, then an error is signalled.
:rehash-threshold	Sets the growth threshold of the table. This is how full the table can become before it must grow. If the value is an integer greater than zero and less than the **:size**, then it specifies the number of entries at which growth occurs. If the value is a floating point number between zero and one, then it specifies the proportion of entries that can be filled before growth occurs. If the value is neither an integer or a floating-point number, then an error is signalled.
:initial-contents	Sets the initial contents for the new table. It can either be a table object to copy the contents from, or a sequence of keys and values to fill the table with.
:hash-function	Specifies the hashing function to be used for the table, if necessary. The default is based on the **:test** predicate, so there is no single default hash function.
:mutating	Turns off the mutation paradigm if you've found that your table doesn't change at some point and you don't want the overhead involved with mutating.

Other keywords that can be passed to the **make-hash-table** function are discussed in a section on extending the table facility. See the section "Extensibility and Additional Features", page 483.

18.3 Table Internals

18.3.1 More About Tables

Many types of table objects are available, but only four basic differences exist among them, namely:

Mutability	You can make a table that changes internal representation at run-time, with the **:mutating** keyword argument to **make-hash-table**. It is usually better for the table to be free to change its internal representation since the representation is picked to be the most efficient for the data currently stored in the table. If you are sure that you have a representation that is efficient, then it might be a good idea to turn off mutation.
Predicate	The predicate is some symbol or function that is called to check the keys. The default predicate is **eql**, however, there are optimizations available for using **eq** and **equal** also. The predicate symbol should be picked very carefully if the default is not used, as some types of tables are optimized for certain predicates. For instance, tables with plist internal representations are optimized for **eq**, and there is a **char-equal** optimization for tables with block array internal representations.
Representation	The internal representation of the table can change if the **:mutating** keyword argument is **t** (the default). The representation changes when the size of the data passes some size threshold, either upward or downward. Some of the representations are hash array, alist and set.
Locking	When the table is accessed for a read or update operation, the data are subject to corruption. This could happen because the table changes representation as another entry is added or because the garbage collector starts up as an entry is being read. To prevent this sort of problem, the table can be locked against interrupts and other processes. Some tables are sensitive to garbage collection; for example, those with an internal representation of hash array which use hash functions based on pointer information. These tables are locked against garbage collection by the table facility automatically.

The facility can be changed and customized beyond the scope of the differences discussed here. See the section "Extensibility and Additional Features", page 483.

18.3.2 Flavors and Tables

The table management facility is based on Flavors, and uses Flavors to define table types and generic functions for accessing the tables. It also defines a large family of table flavors that are built from a base flavor and mixins.

One of the most important mixins in this implementation is the **mutating-table-mixin**. This mixin enables the tables to change internal representation automatically at run-time, based on the initial keywords that you give to **make-hash-table** and the current size and type of the data set being represented. This mixin makes accessing and updating the tables much more efficient. Mutating tables can be optimized for either space or speed, making efficiency constraints more easily controlled.

The mutation works by defining **:after** daemons for the generic functions in the table protocol and uses a **change-instance-flavor** to convert the table to a new flavor returned from the function **find-mutated-flavor**. The generic function, **need-to-mutate-p**, is a predicate defined for the table facility to decide when a table should be mutated.

The mutations that are currently defined are **grow** and **shrink**. These mutations are linked closely to the size of the table, the test being used, and any optimization keywords that are specified when the table is created. Each time the size of the table passes a defined threshold, it changes its internal representation to something more efficient.

The **make-hash-table** function creates a graph in the mutation-map instance variable of each table. The nodes in the graph represent flavors, and the edges represent the various mutations from flavor to flavor. This graph is then used by **find-mutated-flavor** to find the new flavor that is related to the current flavor of the table according to the desired mutation.

As an example, suppose you created a table with the following options:

```
(setq my-table (make-hash-table :size 30 :test 'eq :rehash-size 5))
```

You might get something like this as a result:

```
#<EQ-BLOCK-ARRAY-DUMMY-GC-LOCKING-ASSOCIATION-MUTATING-TABLE 102741253>
```

Then the mutation map for crossing growth thresholds might look something like this:

```
EQ-PLIST-DUMMY-GC-LOCKING-ASSOCIATION-MUTATING-TABLE
        ↓ grow          shrink ↑
EQ-BLOCK-ARRAY-DUMMY-GC-LOCKING-ASSOCIATION-MUTATING-TABLE
        ↓ grow          shrink ↑
EQ-OPEN-SCATTER-GC-LOCKING-HASH-ARRAY-ASSOCIATION-MUTATING-TABLE
```

18.3.3 Extensibility and Additional Features

The table facility can be extended in various ways. One way is to take advantage of some of the extra options to **make-hash-table**, such as **:area** and **:locking**. Another is to add mixins to a flavor of table that is almost what you want.

:area
Similar to those for **make-array** and **make-list**. If **:area** is **nil** (the default), the **default-cons-area** is used. Otherwise, the argument should be the number of the area that you wish to use. For more information on areas: See the section "Areas" in *Internals, Processes, and Storage Management.*

:entry-size
The number of logical pieces for each entry. If the **:entry-size** argument is 1, then the table is of type sequence. In other words, this is a table consisting solely of keys. When adding an entry to the table for this special case, the value associated with the key is ignored. As a style convention, we recommend making the value the same as the key for each entry. For example:

```
(setq plants
  (make-hash-table :size 5 :entry-size 1))

(setf (gethash 'ficus plants) 'ficus)
```

:locking
Specifies the locking strategy for table operations. A number of strategies are available for process, garbage collection, and interrupt locking. The default is to lock against the garbage collector when necessary and to lock against other processes. Locking against GC is necessary when the hash function is sensitive to GC flip.

18.3.4 Hash Primitive

Hashing is a technique used to provide fast retrieval of data in large tables. A function, known as a *hash function*, is created, which takes a key, and produces a number to be associated with that key. This number, or some function of it, can be used to specify where in a table to look for the value associated with the key. It is always possible for two different objects to "hash to the same value"; that is, for the hash function to return the same number for two distinct objects. Good hash functions are designed to minimize this by evenly distributing their results over the range of possible numbers. However, hash table algorithms must still anticipate this problem by providing a secondary search, sometimes known as a *rehash.* For more information, consult a textbook on computer algorithms.

si:equal-hash and **sxhash** provide what is called "hashing on **equal**"; that is, two objects that are **equal** are considered to be "the same" by **si:equal-hash** and **sxhash**. In particular, if two strings differ only in alphabetic case, **si:equal-hash** and **sxhash** return the same object for both of them because they are **equal**. The value returned by **si:equal-hash** and **sxhash** does not depend on the case of any strings. Therefore, **si:equal-hash** and **sxhash** are useful for retrieving data when two keys that are not the same object but are **equal**, are considered the same.

If you consider two such keys to be different, then you need "hashing on **eq** or eql", where two different objects are always considered different. This is done by returning the virtual address of the storage associated with the object. **eq** and **eql** hash tables function properly, even though the address associated with an object can be changed by the relocating garbage collector. When copying, if the garbage collector changes the addresses of object, it lets the hash facility know so that **gethash** rehashes the table based on the new object addresses.

18.4 Other Data Types Used as Tables

Sometimes it is easier or more convenient to make tables without the table management facility. For example, if you have an application which requires a very small table which has a relatively static size or a lookup table which never changes after it's created, you might find that the overhead involved in the table management facility slows down your application too much. For those situations, the traditional tables, made from lists and arrays, are useful. This section covers various types of non-mutating tables.

A number of tools are currently available for creating and using various types of specialized tables. These tools include functions that operate on sequences, association lists, property lists, general lists, arrays and hashed arrays. All of these types of tables accomplish the same thing, that is, storing data in a tabular fashion, but each one has a different, incompatible interface.

18.4.1 Sequences as Tables

The simplest table is one whose entries contain one element. This type of table is a *sequence*, which is either a *set* or a *vector*. A set is a list of items and a vector is a one-dimensional array.

There are functions to add (**concatenate, push**), remove (**delete, remove**), and search for (**find, position**) items in a set. An example of a simple table made up of a sequence might be a set of bird names. Its list representation would be:

```
(heron turkey eagle pelican loon stork)
```

and a more abstract way of thinking about it as a table would be:

KEY
heron
turkey
eagle
pelican
loon
stork
...

A table of this type would quickly become inefficient as more entries were added because sequential searching through the keys would take increasingly longer amounts of time.

Vectors are very similar to sets but they are array structures instead of list structures. Therefore, the functions used to add, remove, and search through the structure are different, even though the general principals remain the same.

The two major difference between vectors and sets are:

- The element access time for any element of a one-dimensional array is constant, whereas the element access time for an element of a set depends on the length of the set and where that element resides in the set.

- Adding a new element to the front of a set takes constant time, whereas the adding a new element to the front of an array takes an amount of time proportional to the length of the array.

For more information on sequences and sequence functions: See the section "Sequences", page 187.

18.4.2 Lists as Tables

You can build more complex tables out of sets and vectors. Tables made from general lists, alists, plists, and arrays are examples of how this can be applied.

A table whose entries each contain two elements is an *association list.* Association lists, or alists, are lists of conses, and are very commonly used for tabular data. The car of each cons is a key and the cdr is a value. This value can be a symbol, a list of associated data, or any other structure. The functions **assoc** and **assq** retrieve the value from an association list, given the key. An example of a table of this type is:

KEY	VALUE
heron	wader
loon	diver
eagle	raptor
...	...

Its alist representation would be

```
((heron . wader) (loon . diver) (eagle . raptor))
```

Given this alist, you could retrieve the class of any bird in the list.

Another type of table with two elements is the *property list* or plist. The main difference between an alist and a plist is the internal representation. An association list is represented by dotted pairs, while a property list is represented as a list of conses, or logical pairs.

For each property in a plist has an *indicator* and a *value*. An example of a property list would be a set of properties that belong to a bird, say an eagle.

INDICATOR	VALUE
color	(brown white)
food	(mice snakes)
activity-period	day
...	...

The representation for this property list would be:

```
(color (brown white) food (mice snakes) activity-period day)
```

You would then say that "the value of the **color** property is the list **(brown white)**." You can retrieve the value of an indicator with the **get** function.

For more information on lists: See the section "Lists", page 131.

18.4.3 Arrays as Tables

If the values you are working with can be stored in a fixed number of rows and columns, arrays can be more efficient than lists. An example of an array might be a table of animal types expanded to include exactly five examples of animals in each of 100 families. It might look like this:

KEY	VALUE1	VALUE2	VALUE3	VALUE4	VALUE5
bird	heron	turkey	eagle	pelican	loon
mammal	cat	dog	monkey	whale	elephant
reptile	python	basilisk	turtle	monitor	crocodile
...	...	...	...	...	...

To pick elements out of arrays, use the **aref** function.

Hashing is a technique which provides fast retrieval of data in large tables. A function, known as a *hash function*, is created. This function specifies where in the table to look for the value associated with a given key. The default hash function is based on the **:test** predicate. Although hashed arrays are useful for making larger tables more efficient, they are too cumbersome for tables with few entries.

For more information on arrays: See the section "Arrays", page 157.

18.4.4 Zetalisp Hash Tables

Zetalisp hash tables are an older implementation of tables being phased out in favor of the table management facility. Keep in mind that although they are still part of Genera, they will be removed at some point in the future, and are now considered obsolete. You should think about changing your current hash tables over to the new facility.

Zetalisp hash tables are implemented as instances of flavors of two types, the difference being whether the keys are compared using **eq** or **equal**.

You can create a new hash table using the predicate **eq** for comparisons of the key with the function **zl:make-hash-table**. You can create a new hash table using the predicate **equal** for comparisons of the key with the function **zl:make-equal-hash-table**.

You can add new entries to a hash table with the **zl:puthash** function. To look up a key and find the associated value, use the **gethash** function. To remove an entry, use **remhash**. Here is a simple example:

```
(setq a (zl:make-hash-table))
   => #<EQ-HASH-TABLE 40053062>

(zl:puthash 'color 'brown a) => BROWN

(zl:puthash 'name 'fred a) => FRED

(gethash 'color a) => BROWN and T

(gethash 'name a) => FRED and T
```

18.4.5 Heaps

A heap is a data structure in which each item is ordered by some predicate (for example, less-than) on its associated key. You can add an item to the heap, delete an item from it, or look at the top item. The **:top** operation is guaranteed to return the first item in the heap. In the less-than example, this would be the smallest item. Heaps are useful in maintaining priority queues.

18.4.5.1 Heap Functions and Methods

:clear	Removes all of the entries from the heap.
:delete-by-item *item* &optional (*equal-predicate* #'=)	Finds the first item whose key satisfies *equal-predicate* and deletes it. The first argument to *equal-predicate* is the current item from the heap and the second argument is *item.*
:delete-by-key *key* &optional (*equal-predicate* #'=)	Finds the first item whose key satisfies *equal-predicate* and deletes it. The first argument to *equal-predicate* is the current key from the heap and the second argument is *key.*
:describe	Gives the predicate, number of elements, and optionally the contents of the heap.
:empty-p	Returns **t** if the heap is empty, otherwise returns **nil.**
:find-by-item *item* &optional (*equal-predicate* #'=)	Finds the first item in the heap that matches *item.*
:find-by-key *key* &optional (*equal-predicate* #'=)	Finds the first item in the heap whose key matches *key.*
:insert *item key*	Inserts *item* into the heap based on *key.*
make-heap	Creates a new heap.
:remove	Removes the top item from the heap.
:top	Returns the value and key of the top item on the heap.

A heap replacement will be provided in a future release. It will be replaced by a new table type in the Table Management facility. In the meantime, however, heaps are useful for keeping ordered tables.

18.5 Converting Zetalisp Hash Tables to Table Objects

This section illustrates how the syntax of the new table management facility differs from the old Zetalisp hash tables. The hash table syntax still works in Genera 7.0, but will be phased out in a future release. It is very straightforward to change over to the new facility.

18.5.1 New Table Objects Versus Old Zetalisp Hash Tables

New applications should make tables with the table management facility's **make-hash-table** function, rather than calling **make-instance**, **zl:make-equal-hash-table**, or **zl:make-hash-table**. For example:

```
Old syntax:     (setq table
                  (make-instance 'si:eq-hash-table :size 20))
Old syntax:     (setq table (make-equal-hash-table :size 20))
Old syntax:     (setq table (zl:make-hash-table :size 20))

New syntax:     (setq table (make-hash-table :size 20))
```

18.5.2 Inserting New Entries

Wherever **:put-hash**, **zl:puthash**, or **zl:puthash-equal** are used, the equivalent **setf** form should be used instead. For example:

```
Old syntax:     (send table ':put-hash 'color 'brown)
Old syntax:     (zl:puthash 'color 'brown table)
Old syntax:     (zl:puthash-equal 'color 'brown table)

New syntax:     (setf (gethash 'color table) 'brown)
```

18.5.3 Generic Functions Versus Table Messages

All instances of messages should be changed to the equivalent generic function. For example:

```
Old syntax:     (send table ':describe)

New syntax:     (describe table)
```

The old messages and their new equivalents follow:

:clear-hash	**clrhash**
:describe	**describe**
:filled-elements	**hash-table-count**
:get-hash	**gethash**
:map-hash	**maphash**
:modify-hash	**modify-hash**
:rem-hash	**remhash**
:swap-hash	**swaphash**

18.5.4 New Table Facility Functions Versus Zetalisp Functions

Zetalisp functions should be changed over to their Genera equivalent. The old functions and their equivalents follow:

zl:clrhash-equal	**clrhash**
zl:gethash	**gethash**
zl:gethash-equal	**gethash**
zl:maphash-equal	**maphash**
zl:remhash-equal	**remhash**
zl:swaphash-equal	**swaphash**

PART V.

Managing Programs

19. Evaluation

19.1 Introduction to Evaluation

The following is a complete description of the actions taken by the evaluator, given a *form* to evaluate.

form	*Result*
A symbol	The binding of *form*. If *form* is unbound, an error is signalled. See the section "Variables", page 494. Some symbols can also be constants, for example: **t**, **nil**, keywords, and **defconstant**.
Not a symbol or list	*form*
A list	The evaluator examines the car of the list to figure out what to do next. There are three possibilities: the form can be a *special form*, a *macro form*, or a *function form*. Conceptually, the evaluator knows specially about all the symbols whose appearance in the car of a form make that form a special form, but the way the evaluator actually works is as follows. If the car of the form is a symbol, the evaluator finds the function definition of the symbol in the local lexical environment. If no definition exists there, the evaluator finds it in the global environment, which is in the function cell of the symbol. In either case, the evaluator starts all over as if that object had been the car of the list. (See the section "Symbols and Keywords", page 123.) If the car is not a symbol, but a list whose car is the symbol **special**, this is a macro form or a special form. If it is a "special function", this is a special form. See the section "Kinds of Functions", page 256. Otherwise, it should be a regular function, and this is a function form.
A special form	It is handled accordingly; each special form works differently. See the section "Kinds of Functions", page 256. The internal workings of special forms are explained in more detail in that section, but this hardly ever affects you.
A macro form	The macro is expanded and the result is evaluated in place of *form*. See the section "Macros", page 285.

A function form It calls for the *application* of a function to *arguments*. The car of the form is a function or the name of a function. The cdr of the form is a list of subforms. Each subform is evaluated, sequentially. The values produced by evaluating the subforms are called the "arguments" to the function. The function is then applied to those arguments. Whatever results the function returns are the values of the original *form*.

See the section "Variables", page 494. The way variables work and the ways in which they are manipulated, including the binding of arguments, is explained in that section. See the section "Evaluating a Function Form", page 504. That section contains a basic explanation of functions. See the section "Multiple Values", page 512. The way functions can return more than one value is explained there. See the section "Functions", page 251. The description of all of the kinds of functions, and the means by which they are manipulated, is there. The **evalhook** facility lets you do something arbitrary whenever the evaluator is invoked. See the section "**evalhook**" in *Program Development Utilities*. Special forms are described throughout the documentation.

19.2 Variables

In Symbolics Common Lisp, variables are implemented using symbols. Symbols are used for many things in the language, such as naming functions, naming special forms, and being keywords; they are also useful to programs written in Lisp, as parts of data structures. But when the evaluator is given a symbol, it treats it as a variable. If it is a special variable, it uses the value cell to hold the value of the variable. If it is not special, it looks it up in the local lexical environment. If you evaluate a symbol, you get back the contents of the symbol's value cell.

19.2.1 Changing the Value of a Variable

There are two different ways of changing the value of a variable. One is to *set* the variable. Setting a variable changes its value to a new Lisp object, and the previous value of the variable is forgotten. Setting of variables is usually done with the **setq** special form.

The other way to change the value of a variable is with *binding* (also called "lambda-binding"). When a variable is bound, its old value is first saved away, and then the value of the variable is made to be the new Lisp object. When the binding is undone, the saved value is restored to be the value of the variable. Bindings are always followed by unbindings. This is enforced by having binding done only by special forms that are defined to bind some variables, then evaluate some subforms, and then unbind those variables. So the variables are all unbound when the form is finished. This means that the evaluation of the form does not disturb the values of the variables that are bound; their old value, before the evaluation of the form, is restored when the evaluation of the form is completed. If such a form is exited by a nonlocal exit of any kind, such as **throw** or **return**, the bindings are undone whenever the form is exited.

19.2.2 Binding Variables

The simplest construct for binding variables is the **let** special form. The **do** and **prog** special forms can also bind variables, in the same way **let** does, but they also control the flow of the program and so are explained elsewhere. See the section "Iteration", page 531. **let*** is just a sequential version of **let**.

Binding is an important part of the process of applying functions to arguments. See the section "Evaluating a Function Form", page 504.

19.2.3 Kinds of Variables

In Symbolics Common Lisp, there are three kinds of variables: *local*, *special*, and *instance*. A special variable has dynamic scope: any Lisp expression can access it simply by mentioning its name. A local variable has lexical scope: only Lisp expressions lexically contained in the special form that binds the local variable can access it. An instance variable has a different kind of lexical scope: only Lisp expressions lexically contained in methods of the appropriate flavor can access it. Instance variables are explained in another section. See the section "Overview of Flavors", page 47.

Variables are assumed to be local unless they have been declared to be special or they have been implicitly declared to be instance variables by **defmethod**. Variables can be declared special by the special forms **defvar** and **zl:defconst**, or by explicit declarations. See the section "Declarations", page 260. The most common use of special variables is as "global" variables: variables used by many different functions throughout a program, that have top-level values. Named constants are considered to be a kind of special variable whose value is never changed.

When a Lisp function is compiled, the compiler understands the use of symbols as variables. However, the compiled code generated by the compiler does not actually use symbols to represent nonspecial variables. Rather, the compiler converts the references to such variables within the program into more efficient references that do not involve symbols at all. The interpreter stores the values of variables in the same places as the compiler, but uses less specialized and efficient mechanisms to access them.

The value of a special variable is stored in the value cell of the associated symbol. Binding a special variable saves the old value away and then uses the value cell of the symbol to hold the new value.

When a local variable is bound, a memory cell is allocated in a hidden, internal place (the Lisp control stack) and the value of the variable is stored in this cell. You cannot use a local variable without first binding it; you can only use a local variable inside a special form that binds that variable. Local variables do not have any "top-level" value; they do not even exist outside the form that binds them.

The value of an instance variable is stored in an instance of the appropriate flavor. Each instance has its own copy of the instance variable. It is impermissible to bind an instance variable.

Local variables and special variables do not behave quite the same way, because "binding" means different things for the two of them. Binding a special variable saves the old value away and then uses the value cell of the symbol to hold the new value. Binding a local variable, however, does not do anything to the symbol. In fact, it creates a new memory cell to hold the value, that is, a new local variable.

A reference to a variable that you did not bind yourself is called a *free reference.* When one function definition is nested inside another function definition, using **lambda**, **flet**, or **labels**, the inner function has access to the local variables bound by the outer function. An access by the inner function to a local variable of the outer function looks like a free reference when only the inner function is considered. However, when the entire surrounding context is considered, it is a bound reference. We call this a *captured free reference.* When a function definition is nested inside a method, it can refer to instance variables just as the method can.

You cannot use a local variable without first binding it. Another way to say this is that you cannot ever have an uncaptured free reference to a local variable. If you try to do so, the compiler complains and assumes that the variable is special, but was accidentally not declared. The interpreter also assumes that the variable is special, but does not print a warning message.

Here is an example of how the compiler and the interpreter produce the same results, but the compiler prints more warning messages.

```
(setq a 2)          ;Set the special variable a to the value 2.
                    ;But don't declare a special.

(defun foo ()       ;Define a function named foo.
  (let ((a 5))      ;Bind the local variable a to the value 5.
    (bar)))         ;Call the function bar.

(defun bar ()       ;Define a function named bar.
  a)                ;It makes a free reference to the special variable a.
```

```
(foo) => 2           ;Calling foo returns 2.

(compile 'foo)        ;Now compile foo.
                      ;This warns that the local variable a was bound,
                      ;but was never used.

(foo) => 2           ;Calling foo still returns 2.

(compile 'bar)        ;This warns about the free reference to a.

(foo) => 2           ;Calling foo still returns 2.
```

When **bar** was compiled, the compiler saw the free reference and printed a warning message: `Warning: a declared special.` It automatically declared **a** to be special and proceeded with the compilation. It knows that free references mean that special declarations are needed. But when a function, such as **foo** in the example, is compiled that binds a variable that you want to be treated as a special variable but that you have not explicitly declared, there is, in general, no way for the compiler to automatically detect what has happened, and it produces incorrect output. So you must always provide declarations for all variables that you want to be treated as special variables.

When you declare a variable to be special using **defvar** rather than **declare** inside the body of a form, the declaration is "global"; that is, it applies wherever that variable name is seen. After **fuzz** has been declared special using **defvar**, all following uses of **fuzz** are treated as references to the same special variable. Such variables are called "global variables", because any function can use them; their scope is not limited to one function. The special forms **defvar** and **zl:defconst** are useful for creating global variables; not only do they declare the variable special, but they also provide a place to specify its initial value, and a place to add documentation. In addition, since the names of these special forms start with "**def**" and since they are used at the top level of files, the editor can find them easily.

19.2.4 Standard Variables

Standard variables are special variables that are used to control some aspect of the Lisp environment. Their initial (standard) values are stored in **si:*standard-bindings***. If something binds one of the standard variables, the binding is stored in **si:*interactive-bindings***. **si:*standard-bindings*** and **si:*interactive-bindings***. **si:*interactive-bindings*** is never set, only bound and **si:*standard-bindings*** is never bound, only set.

When a breakpoint of some kind is entered, the system finds out the standard values for all the symbols defined with **sys:defvar-standard**. It then compares these values against the current bindings for these symbols. If the current bindings do not match the standard bindings, you are warned, and the symbols are bound to the standard values. The standard binding for a variable is gotten by looking on **si:*interactive-bindings***. If no binding is found on **si:*interactive-bindings***, then **si:*standard-bindings*** is checked. For example, **zwei:com-break** puts the value of ***package***, ***read-base***, and ***print-base*** from the file attribute list onto **si:*interactive-bindings*** so that they become the standard binding for Zmacs. **zl:pkg-goto** puts the new value of ***package*** onto **si:*standard-bindings***. Evaluation of forms in Zmacs, for example, Evaluate Into Buffer m-X, also binds the symbols to their standard values.

As a result, whenever you enter a breakpoint you are guaranteed predictable, consistent behavior with regard to the bindings of these variables.

These are the currently defined standard variables, their standard values.

symbol	*standard value*
gprint:*inspecting*	**nil**
cp:*command-table*	User Command Table
neti:*inhibit-obsolete-information-warning*	**t**
package	**common-lisp-user**
read-suppress	**nil**
read-default-float-format	**single-float**
print-pretty-printer	**gprint:print-object**
print-structure-contents	**t**
print-bit-vector-length	**nil**
print-string-length	**nil**
print-array-length	**nil**
print-readably	**nil**
print-array	**nil**
print-gensym	**t**
print-case	**:upcase**
print-length	**nil**
print-level	**nil**
print-circle	**nil**
print-base	**10**
print-radix	**nil**
print-abbreviate-quote	**nil**
print-pretty	**t**
print-escape	**t**
sys:default-cons-area	**4**
readtable	Common-Lisp Readtable
read-base	**10**
prinl	**nil**

Notes:

1. The value of ***package*** must be an unlocked package in **si:*reasonable-packages*** that uses one of the packages in **si:*reasonable-packages***.

2. The ***readtable*** must be one of the readtables on the list **si:*valid-readtables***.

3. The value of **sys:default-cons-area** must be an allocated area.

The following functions and variables pertain to standard variables:

sys:defvar-standard *var initial-value*	Defines a *standard value* that the variable should be bound to in command and breakpoint loops.
sys:standard-value-p *symbol*	Returns **t** if *symbol* has a standard value.
sys:standard-value *symbol*	Returns the standard value of *symbol.*
(setf (sys:standard-value *symbol***))**	Changes the standard value of *symbol.*
zl:setq-standard-value *name form*	Sets the standard value of *name* to the value of *form.*

Standard variables are particularly useful in command loops. The following functions are useful for writing your own Lisp style command loops.

sys:standard-value-let *vars-and-vals* &body *body*	Like **let** except that it pushes the values in *vals* onto the **si:*interactive-bindings***, causing them to become standard values.
sys:standard-value-let* *vars-and-vals* &body *body*	Like **let*** except that it pushes the values in *vals* onto the **si:*interactive-bindings***.
sys:standard-value-progv *vars-and-vals* &body *body*	Causes all of the symbols in *vars* to have their corresponding value in *vals* pushed onto the **si:*interactive-bindings***.

si:standard-readtable *Variable*

The value of **si:standard-readtable** is that **zl:readtable** to use when typing forms interactively to the Lisp interpreter. When a distribution world is cold booted, the value of **si:standard-readtable** is a copy of **si:initial-readtable**. If you wish to customize the syntax of forms typed to the Lisp interpreter, you should make your customizations to **si:standard-readtable**. **zl:readtable** is bound to **si:standard-readtable** whenever a break loop or debug loop is entered. **zl:readtable** is set to **si:standard-readtable** using the standard variable mechanism whenever the machine is warm booted.

If warm booting the machine were impossible, then **si:standard-readtable** would not be necessary. The top-level value of **zl:readtable** could be used instead. However, if the machine is warm booted while **zl:readtable** is bound, the top-level value of **zl:readtable** is lost.

Examples:

- This example illustrates the use of binding **zl:readtable** in order to implement a special syntax. Forms are to be read from a file while preserving the case of symbols.

```
(defvar case-sensitive-readtable (copy-readtable))

(loop for code from (char-code #/a) to (char-code #/z)
      as char = (code-char code)
      do (setf (si:rdtbl-trans case-sensitive-readtable code) char))

(defun read-case-sensitive-file (file)
  (with-open-file (stream file :direction :input)
    (let ((readtable case-sensitive-readtable))
      (loop do (process-form (read stream))))))
```

  In case an error occurs while inside **process-form** or inside a reader macro invoked by **zl:read, zl:readtable** is bound to **si:standard-readtable,** which is most useful for debugging.

- This example illustrates the use of **si:standard-readtable** and **si:initial-readtable** to customize the environment for typing expressions interactively. "@" is defined as an abbreviation for **location-contents**, in the same manner that "'" is an abbreviation for **quote**.

```
(defun at-sign-macro (ignore stream)
  (values (list 'location-contents (read stream)) 'list))

(defvar my-readtable (copy-readtable))
(set-syntax-macro-char #/@ 'at-sign-macro my-readtable)
```

```
(defun enable-my-readtable ()
  (setq si:standard-readtable my-readtable)
  (setq readtable my-readtable))

(defun disable-my-readtable ()
  (setq si:standard-readtable si:initial-readtable)
  (setq readtable si:initial-readtable))
```

While it is useful for the user to set the values of **zl:readtable** and **si:standard-readtable**, the value of **si:initial-readtable** should never be changed. In addition, the **zl:readtable** that is the value of **si:initial-readtable** should never be modified, modifications should be made only to the **zl:readtable** that is the value of **si:standard-readtable**. See the function **zl:copy-readtable** in *Reference Guide to Streams, Files, and I/O.*

See the section "The Readtable" in *Reference Guide to Streams, Files, and I/O.*

19.2.5 Special Forms for Setting Variables

setf *reference value*	Takes a form that *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing.
psetf *variable-value-pairs*	Like **zl:setf** but performs all the updates in *parallel*, that is, simultaneously.
setq *variable value*	Sets *variable* to *value*.
psetq *variable-value-pairs*	Like **setq** but performs all the assignments in *parallel*, that is, simultaneously.
zl:psetq *variable value*	Sets *variable* to *value* after evaluating *value*.

19.2.6 Special Forms for Binding Variables

let *((var value)...) body*	Binds some variables to some objects, and evaluates *body* in the context of those bindings.
let* *((var value)...) body...*	Like **let**, except that the binding is sequential.
compiler-let *bindlist body...*	Like **let** with variables declared special when interpreted. For compiled code, compiles the body with the bindings specified by *bindlist* in effect.
letf *places-and-values body...*	Like **let**, except that it can bind any storage cells not just variables.
letf* *places-and-values body...*	Like **let**, except that it does the binding sequentially.

let-if *condition ((var value)...) body...*	Like **let** except the binding of variables is conditional.
let-globally *((var value)...) body...*	Saves the old values and *sets* the variables, setting up an **unwind-protect**.
let-globally-if *predicate varlist body...*	Binds the variables only if *predicate* evaluates to something other than **nil**.
zl:progv *vars vals* &body *body*	Binds *vars* to *vals* and evaluates *body*. *vars* and *vals* are computed quantities.
progw *vars-and-vals* &body *body*	Like **zl:progv** except the evaluation is sequential.
destructuring-bind *pattern datum* &body *body*	Binds variables to values, using **defmacro**'s destructuring facilities, and evaluates the body forms in the context of those bindings.
zl:desetq	Lets you assign values to variables through destructuring patterns.
zl:dlet	Binds variables to values, using destructuring, and evaluates the body forms in the context of those bindings. The bindings happen in parallel.
zl:dlet*	Like **zl:dlet** except the bindings happen sequentially.

19.2.7 Special Forms for Defining Special Variables

defvar *var initial-value*	Declares *var* to be a global variables. **defvar** should be used only at top level in a program, never in a function definition.
sys:defvar-resettable *var initial-value warm-boot-value*	Like **defvar**, except that it also accepts a *warm-boot value.*
defconstant *variable initial-value*	Declares the use of a named constant in a program.
defparameter *variable initial-value*	The same as **defvar**, except that *variable* is always set to *initial-value* regardless of whether *variable* is already bound.
zl:defconst *variable initial-value*	The same as **defvar**, except that *variable* is always set to *initial-value* regardless of whether *variable* is already bound.

19.3 Generalized Variables

In Lisp, a variable is something that can remember one piece of data. The main operations on a variable are to recover that piece of data, and to change it. These might be called *access* and *update.* The concept of variables named by symbols can be generalized to any storage location that can remember one piece of data, no matter how that location is named. See the section "Variables", page 494.

For each kind of generalized variable, there are typically two functions that implement the conceptual *access* and *update* operations. For example, **symbol-value** accesses a symbol's value cell, and **set** updates it. **array-leader** accesses the contents of an array leader element, and **zl:store-array-leader** updates it. **car** accesses the car of a cons, and **rplaca** updates it.

Rather than thinking in terms of two functions that operate on a storage location somehow deduced from their arguments, we can shift our point of view and think of the access function as a *name* for the storage location. Thus **(symbol-value 'foo)** is a name for the value of **foo**, and **(aref a 105)** is a name for the 105th element of the array **a**. Rather than having to remember the update function associated with each access function, we adopt a uniform way of updating storage locations named in this way, using the **setf** special form. This is analogous to the way we use the **setq** special form to convert the name of a variable (which is also a form that accesses it) into a form that updates it.

setf is particularly useful in combination with structure accessors, such as those created with **defstruct**, because the knowledge of the representation of the structure is embedded inside the accessor, and you should not have to know what it is in order to alter an element of the structure.

setf is actually a macro that expands into the appropriate update function. It has a database that associates from access functions to update functions.

setf	Takes a form that *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing.

Besides the *access* and *update* conceptual operations on variables, there is a third basic operation, which we might call *locate.* Given the name of a storage cell, the *locate* operation returns the address of that cell as a locative pointer. See the section "Cells and Locatives", page 29. locative pointer is a kind of name for the variable that is a first-class Lisp data object. It can be passed as an argument to a function that operates on any kind of variable, regardless of how it is named. It can be used to *bind* the variable, using the **zl:bind** subprimitive.

Of course this can only work on variables whose implementation is really to store their value in a memory cell. A variable with an *update* operation that encrypts the value and an *access* operation that decrypts it could not have the *locate* operation, since the value as such is not directly stored anywhere.

locf	Takes a form that *accesses* some cell and produces a corresponding form to create a locative pointer to that cell.

Both **setf** and **locf** work by means of property lists. When the form **(setf (aref q 2) 56)** is expanded, **setf** looks for the **setf** property of the symbol **aref**. The value of the **setf** property of a symbol should be a cons whose car is a pattern to be matched with the *access-form*, and whose cdr is the corresponding *update-form*, with the symbol **si:val** in place of the value to be stored. The **setf** property of **aref** is a cons whose car is **(aref array . subscripts)** and whose cdr is **(zl:aset si:val array . subscripts)**. If the transformation that **setf** is to do cannot be expressed as a simple pattern, an arbitrary function can be used: When the form **(setf (foo bar) baz)** is being expanded, if the **setf** property of **foo** is a symbol, the function definition of that symbol is applied to two arguments, **(foo bar)** and **baz**, and the result is taken to be the expansion of the **setf**.

Similarly, the **locf** function uses the **locf** property, whose value is analogous. For example, the **locf** property of **aref** is a cons whose car is **(aref array . subscripts)** and whose cdr is **(zl:aloc array . subscripts)**. There is no **si:val** in the case of **locf**.

incf	Increments the value of a generalized variable.
decf	Decrements the value of a generalized variable.
zl:swapf	Exchanges the value of one generalized variable with that of another.

19.4 Evaluating a Function Form

Evaluation of a function form works by applying the function to the results of evaluating the argument subforms. What is a function, and what does it mean to apply it? Symbolics Common Lisp contains many kinds of functions, and applying them can do many different kinds of things. This section explains the most basic kinds of functions and how they work, and in particular, *lambda lists* and all their important features.

The simplest kind of user-defined function is the *lambda-expression*, which is a list that looks like:

(lambda *lambda-list body1 body2*...)

The first element of the lambda-expression is the symbol **lambda**; the second element is a list called the *lambda list*, and the rest of the elements are called the *body*. The lambda list, in its simplest form, is just a list of variables. Assuming that this simple form is being used, here is what happens when a lambda-expression is applied to some arguments.

1. The number of arguments and the number of variables in the lambda list must be the same, or else an error is signalled.

2. Each variable is bound to the corresponding argument value.

3. The forms of the body are evaluated sequentially.

4. The bindings are all undone and the value of the last form in the body is returned.

This might sound something like the description of **let**. The most important difference is that the lambda-expression is a *function*, not a form. A **let** form gets evaluated, and the values to which the variables are bound come from the evaluation of some subforms inside the **let** form; a lambda-expression gets applied, and the values are the arguments to which it is applied.

The variables in the lambda list are sometimes called *parameters*, by analogy with other languages. Some other terminologies refer to these as *formal parameters*, and to arguments as *actual parameters*.

Lambda lists can have more complex structure than simply being a list of variables. Additional features are accessible by using certain keywords (which start with **&**) and/or lists as elements of the lambda list.

The principal weakness of the simple lambda lists is that any function written with one must only take a certain fixed number of arguments. As we know, many very useful functions, such as **list**, **append**, +, and so on, accept a varying number of arguments. Maclisp solved this problem by the use of *lexprs* and *lsubrs*, which were somewhat inelegant since the parameters had to be referred to by numbers instead of names (for example, **(zl:arg 3)**). (For compatibility reasons, Symbolics Common Lisp supports *lexprs*, but they should not be used in new programs). Simple lambda lists also require that arguments be matched with parameters by their position in the sequence. This makes calls hard to read when there are a great many arguments. Keyword parameters enable the use of other styles of call which are more readable.

In general, a function in Symbolics Common Lisp has zero or more *positional* parameters, followed if desired by a single *rest* parameter, followed by zero or more *keyword* parameters. The positional parameters can be *required* or *optional*, but all the optional parameters must follow all the required ones. The required/optional distinction does not apply to the rest parameter.

Keyword parameters are always optional, regardless of whether the lambda list contains **&optional**. Any **&optional** appearing after the first keyword argument has no effect. **&key** and **&rest** are independent. They can both appear and they both use the same arguments from the argument list. The only rule is that **&rest** must appear before **&key** in the lambda list.

This is the ordering rule for lambda-list keywords. The following keywords must appear in this order, any or all of them can be omitted, and they cannot appear multiple times:

```
&optional &rest &key &allow-other-keys &aux
```

There are some other keywords in addition to those mentioned here. See the section "Lambda-List Keywords", page 260.

The caller must provide enough arguments so that each of the required parameters gets bound, but extra arguments can be provided for some of the optional parameters. Also, if there is a rest parameter, as many extra arguments can be provided as desired, and the rest parameter is bound to a list of all these extras. Optional parameters can have a *default-form*, which is a form to be evaluated to produce the default value for the parameter if no argument is supplied.

Positional parameters are matched with arguments by the position of the arguments in the argument list. Keyword parameters are matched with their arguments by matching the keyword name; the arguments need not appear in the same order as the parameters. If an optional positional argument is omitted, no further arguments can be present. Keyword parameters allow the caller to decide independently for each one whether to specify it. If a keyword is duplicated among the keyword arguments, the leftmost occurrence of the keyword takes precedence.

19.4.1 Binding Parameters to Arguments

When **zl:apply** (the primitive function that applies functions to arguments) matches up the arguments with the parameters, it follows this algorithm:

1. The positional parameters are dealt with first.

2. The first required positional parameter is bound to the first argument. **zl:apply** continues to bind successive required positional parameters to the successive arguments. If, during this process, there are no arguments left but some required positional parameters remain that have not been bound yet, it is an error ("too few arguments").

3. After all required parameters are handled, **zl:apply** continues with the optional positional parameters, if any. It binds successive parameters to the next argument. If, during this process, there are no arguments left, each remaining optional parameter's default-form is evaluated, and the parameter is bound to it. This is done one parameter at a time; that is, first one default-form is evaluated, and then the parameter is bound to it, then the next default-form is evaluated, and so on. This allows the default for an argument to depend on the previous argument.

4. If there are no remaining parameters (rest or keyword), and there are no remaining arguments, we are finished. If there are no more parameters but some arguments still remain, an error is signalled ("too many arguments"). If parameters remain, all the remaining arguments are used for both the rest parameter, if any, and the keyword parameters.

   a. First, if there is a rest parameter, it is bound to a list of all the remaining arguments. If there are no remaining arguments, it gets bound to **nil**.

   b. If there are keyword parameters, the same remaining arguments are used to bind them.

5. The arguments for the keyword parameters are treated as a list of alternating keyword symbols and associated values. Each symbol is matched with the keyword parameter names, and the matching keyword parameter is bound to the value that follows the symbol. All the remaining arguments are treated in this way. Since the arguments are usually obtained by evaluation, those arguments that are keyword symbols are typically quoted in the call; however, they do not have to be. The keyword symbols are compared by means of **eq**, which means they must be specified in the correct package. The keyword symbol for a parameter has the same print name as the parameter, but resides in the keyword package regardless of what package the parameter name itself resides in. (You can specify the keyword symbol explicitly in the lambda list if you must.)

   If any keyword parameter has not received a value when all the arguments have been processed, the default-form for the parameter is evaluated and the parameter is bound to its value. The default form can depend on parameters to its left in the lambda-list.

   There might be a keyword symbol among the arguments that does not match any keyword parameter name. An error is signalled unless **&allow-other-keys** is present in the lambda list, or there is a keyword argument pair whose keyword is **:allow-other-keys** and whose value is not **nil**. If an error is not signalled, then the nonmatching symbols and their associated values are ignored. The function can access these symbols and values through the rest parameter, if there is one. It is common for a function to check only for certain keywords, and pass its rest parameter to another function using **zl:lexpr-funcall**; then that function checks for the keywords that concern it.

The way you express which parameters are required, optional, and rest is by means of specially recognized symbols, which are called *&-keywords*, in the lambda list. All such symbols' print names begin with the character "&". A list of all such symbols is the value of the symbol **lambda-list-keywords**.

19.4.2 Examples of Simple Lambda Lists

The keywords used here are **&key**, **&optional** and **&rest**. The way they are used is best explained by means of examples; the following are typical lambda lists, followed by descriptions of which parameters are positional, rest, or keyword, and those that are required or optional.

`(a b c)`	**a**, **b**, and **c** are all required and positional. The function must be passed three arguments.
`(a b &optional c)`	**a** and **b** are required, **c** is optional. All three are positional. The function can be passed either two or three arguments.
`(&optional a b c)`	**a**, **b**, and **c** are all optional and positional. The function can be passed any number of arguments between zero and three, inclusive.
`(&rest a)`	**a** is a rest parameter. The function can be passed any number of arguments.

`(a b &optional c d &rest e)`

a and **b** are required positional, **c** and **d** are optional positional, and **e** is rest. The function can be passed two or more arguments.

`(&key a b)`

a and **b** are both keyword parameters. A typical call looks like

```
(foo :b 69 :a '(some elements))
```

This illustrates that the parameters can be matched in either order.

`(x &optional y &rest z &key a b)`

x is required positional, **y** is optional positional, **z** is rest, and **a** and **b** are keywords. One or more arguments are allowed. One or two arguments specify only the positional parameters. Arguments beyond the second specify both the rest parameter and the keyword parameters, so that

```
(foo 1 2 :b '(a list))
```

specifies **1** for **x**, **2** for **y**, **(:b (a list))** for **z**, and **(a list)** for **b**. It does not specify **a**.

`(&rest z &key a b c &allow-other-keys)`

z is rest, and **a**, **b** and **c** are keyword parameters. **&allow-other-keys** says that absolutely any keyword symbols can appear among the arguments; these symbols and the values that follow them have no effect on the keyword parameters, but do become part of the value of **z**.

19.4.3 Specifying Default Forms in Lambda Lists

If not specified, the *default-form* for each optional or keyword parameter is **nil**. To specify your own default forms, instead of putting a symbol as the element of a lambda list, put in a list whose first element is the symbol (the parameter itself) and whose second element is the default-form. Only optional and keyword parameters can have default forms; required parameters are never defaulted, and rest parameters always default to **nil**. For example:

```
(a &optional (b 3))
```

The default-form for **b** is **3**. **a** is a required parameter, and so it doesn't have a default form.

```
(&optional (a 'foo) &rest d &key b (c (symeval a)))
```

a's default-form is **'foo**, **b**'s is **nil**, and **c**'s is **(symeval a)**. Note that if the function whose lambda list this is were called with no arguments, **a** would be bound to the symbol **foo**, and **c** would be bound to the binding of the symbol **foo**; this illustrates the fact that each variable is bound immediately after its default-form is evaluated, and so later default-forms can take advantage of earlier parameters in the lambda list. **b** and **d** would be bound to **nil**.

Occasionally it is important to know whether or not a certain optional or keyword parameter was defaulted. You cannot tell from just examining its value, since if the value is the default value, there is no way to tell whether the caller passed that value explicitly, or whether the caller did not pass any value and the parameter was defaulted. The way to tell for sure is to put a third element into the list: the third element should be a variable (a symbol), and that variable is bound to **nil** if the parameter was not passed by the caller (and so was defaulted), or **t** if the parameter was passed. The new variable is called a *supplied-p* variable; it is bound to **t** if the parameter is supplied.

For example:

```
(a &optional (b 3 c))
```

The default-form for **b** is **3**, and the supplied-p variable for **b** is **c**. If the function is called with one argument, **b** is bound to **3** and **c** is bound to **nil**. If the function is called with two arguments, **b** is bound to the value that was passed by the caller (which might be **3**), and **c** is bound to **t**.

`(&key a (b (1+ a) c))`

This is the same as the example above, except that it demonstrates use of a supplied-p variable for a keyword parameter. This example also shows the default value of one keyword parameter depending on a previous keyword parameter.

19.4.4 Specifying a Keyword Parameter's Symbol in Lambda Lists

It is possible to specify a keyword parameter's symbol independently of its parameter name. To do this, use *two* nested lists to specify the parameter. The outer list is the one that can contain the default-form and supplied-p variable. The first element of this list, instead of a symbol, is again a list, whose elements are the keyword symbol and the parameter variable name. For example:

`(&key ((:a a)) ((:b b) t))`

This is equivalent to **(&key a (b t))**.

`(&key ((:base base-value)))`

This allows a keyword that the caller knows under the name **:base**, without making the parameter shadow the value of **zl:base**, which is used for printing numbers.

`(&key ((:base base-value) 10 base-supplied))`

When the **zl:base** keyword is supplied, the default value of 10 is ignored and **zl-user:base-supplied** is bound to **t**. If the keyword is not supplied, **zl-user:base-value** is bound to 10 and **zl-user:base-supplied** is bound to **nil**.

19.4.5 Specifying Aux-variables in Lambda Lists

It is also possible to include in the lambda list some other symbols that are bound to the values of their default-forms upon entry to the function. These are not parameters, and they are never bound to arguments; they just get bound, as if they appeared in a **let*** form. (Whether you use these aux-variables or bind the variables with **let*** is a stylistic decision.)

To include such symbols, put them after any parameters, preceded by the &-keyword **&aux**. For example:

```
(a &optional b &rest c &aux d (e 5) (f (cons a e)))
```

d, **e**, and **f** are bound, when the function is called, to **nil**, **5**, and a cons of the first argument and **5**. Note that aux-variables are bound sequentially rather than in parallel.

19.4.6 Safety Of &rest Arguments

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied, as you must always assume that it is one of these special lists. See the function **zl:copylist** in *Symbolics Common Lisp: Language Dictionary.*

The system does not detect the error of omitting to copy a rest-argument; you simply find that you have a value that seems to change behind your back. At other times the rest-args list is an argument that was given to **zl:apply**; therefore it is not safe to **rplaca** this list, as you might modify permanent data structure. An attempt to **rplacd** a rest-args list is unsafe in this case, while in the first case it signals an error, since lists in the stack are impossible to **rplacd.**

19.5 Some Functions and Special Forms

19.5.1 Function for Evaluation

eval *form*	Evaluates *form*, and returns the result.

19.5.2 Functions for Function Invocation

apply *function* &rest *args*	Applies the function *function* to the
zl:apply *function args*	Applies the function *function* to the list of arguments *args.*
funcall *fn* &rest *args*	Applies the function *fn* to *args.*
zl:lexpr-funcall	Like **apply.**
send	Sends the message named *message-name* to the *object.*
lexpr-send	Sends the message named *message-name* to the *object.*
zl:call	Offers a very general way of controlling what arguments you pass to a function.

19.5.3 Functions and Special Forms for Constant Values

quote *object*	Returns *object.* It is useful specifically because *object* is not evaluated.

function *fn*	The functional interpretation of *fn*.
lambda *fn*	Provided, as a convenience, to obviate the need for using the **function** special form when the latter is used to name an anonymous (lambda) function.
false	Takes no arguments and returns **nil**.
true	Takes no arguments and returns **t**.
ignore	Takes any number of arguments and returns **nil**.
constantp *object*	Returns **t** if *object*, when considered as a form to be evaluated, always evaluates to the same thing.
zl:comment *form*	Ignores its form and returns the symbol **zl:comment**.

19.5.4 Special Forms for Sequencing

progn *body*	The *body* forms are evaluated in order from left to right and the value of the last one is returned.
prog1 *value*	Similar to **progn**, but it returns *value* (its *first* form) rather than its last.
prog2	**prog2** is similar to **progn** and **prog1**, but it returns its *second* form. It is included largely for compatibility with old programs.

19.5.5 Functions for Compatibility with Maclisp Lexprs

zl:arg	**(zl:arg nil)**, when evaluated during the application of a lexpr, gives the number of arguments supplied to that lexpr.
zl:setarg *i x*	Used only during the application of a lexpr. **(setarg** *i x***)** sets the lexpr's *i*'th argument to *x*
zl:listify *n*	Manufactures a list of *n* of the arguments of a lexpr.

19.6 Multiple Values

The Symbolics Lisp Machine includes a facility by which the evaluation of a form can produce more than one value. When a function needs to return more than one result to its caller, multiple values are a cleaner way of doing this than returning a list of the values or **setq**'ing special variables to the extra values. In most Lisp function calls, multiple values are not used. Special syntax is required both to *produce* multiple values and to *receive* them.

19.6.1 Primitive for Producing Multiple Values

The primitive for producing multiple values is **values**, which takes any number of arguments and returns that many values. If the last form in the body of a function is a **values** with three arguments, then a call to that function returns three values. Many system functions produce multiple values, but they all do it via the **values** primitive.

values	Returns multiple values, its arguments.
values-list *list*	Returns multiple values, the elements of the *list.*

19.6.2 Special Forms for Receiving Multiple Values

The special forms for receiving multiple values are **zl:multiple-value**, **multiple-value-bind**, **multiple-value-list**, **multiple-value-call**, and **multiple-value-prog1**. These consist of a form and an indication of where to put the values returned by that form. With the first two of these, the caller requests a certain number of returned values. If fewer values are returned than the number requested, then it is exactly as if the rest of the values were present and had the value **nil**. If too many values are returned, the rest of the values are ignored. This has the advantage that you do not have to pay attention to extra values if you don't care about them, but it has the disadvantage that error-checking similar to that done for function calling is not present.

zl:multiple-value *(variable...) form*	Calls a function that is expected to return more than one value.
multiple-value-bind *(variable...) form body...*	Similar to **zl:multiple-value**, but locally binds the variables that receive the values, rather than setting them, and has a body.
multiple-value-list *form*	Evaluates *form* and returns a list of the values it returned.
multiple-value-call *function body...*	First evaluates *function* to obtain a function. It then evaluates all the forms in *body.*
multiple-value-prog1 *first-form body...*	Like **prog1**, except that if its first form returns multiple values, **multiple-value-prog1** returns those values.

19.6.3 Passing-back of Multiple Values

Due to the syntactic structure of Lisp, it is often the case that the value of a certain form is the value of a subform of it. For example, the value of a **cond** is the value of the last form in the selected clause. In most such cases, if the subform produces multiple values, the original form also produces all of those values. This *passing-back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached. The exact rule governing passing-back of multiple values is as follows:

If *X* is a form, and *Y* is a subform of *X*, then if the value of *Y* is unconditionally returned as the value of *X*, with no intervening computation, then all the multiple values returned by *Y* are returned by *X*. In all other cases, multiple values or only single values can be returned at the discretion of the implementation; users should not depend on whatever way it happens to work, as it might change in the future or in other implementations. The reason we do not guarantee nontransmission of multiple values is because such a guarantee is not very useful and the efficiency cost of enforcing it is high. Even **setq**'ing a variable to the result of a form, then returning the value of that variable might be made to pass multiple values by an optimizing compiler that realized that the **setq**ing of the variable was unnecessary.

Note that use of a form as an argument to a function never receives multiple values from that form. That is, if the form **(foo (bar))** is evaluated and the call to **bar** returns many values, **foo** is still only called on one argument (namely, the first value returned), rather than called on all the values returned. We choose not to generate several separate arguments from the several values, because this makes the source code obscure; it is not syntactically obvious that a single form does not correspond to a single argument. Instead, the first value of a form is used as the argument and the remaining values are discarded. Receiving of multiple values is done only with the special forms discussed in another section. See the section "Special Forms for Receiving Multiple Values", page 513.

19.6.4 Interaction of Some Common Special Forms with Multiple Values

The interaction of special forms with multiple values can be deduced from the rule mentioned in another section: See the section "Passing-back of Multiple Values", page 514. Note well that when it says that multiple values are not returned, it really means that they might or might not be returned, and you should not write any programs that depend on which way it works.

- The body of a **defun** or a **lambda**, and variations such as the body of a function, the body of a **let**, and so on, pass back multiple values from the last form in the body.

- **eval**, **zl:apply**, **funcall**, and **zl:lexpr-funcall** pass back multiple values from the function called. Example:

```
(apply #'floor '(3.4)) => 3 and 0.4000001
```

- **progn** passes back multiple values from its last form. **zl:progv** and **progw** do so also. **prog1** and **prog2**, however, do not pass back multiple values (though **multiple-value-prog1** does).
  Examples:

  ```
  (progn (values 1 2)
         (values 3 4)) => 3 and 4

  (prog1 (values 1 2)
         (values 3 4)) => 1
  ```

- Multiple values are passed back from the last subform of an **and** or **or** form, but not from previous forms since the return is conditional. Remember that multiple values are only passed back when the value of a subform is unconditionally returned from the containing form. For example, consider the form **(or (foo) (bar))**. If **foo** returns a non-**nil** first value, then only that value is returned as the value of the form. But if it returns **nil** (as its first value), then **or** returns whatever values the call to **bar** returns.
  Examples:

  ```
  (or (numberp 'x) (values nil 4 5 6) (values 3 4)) => 3 and 4
  (or (numberp 'x) (values 1 2) (values 3 4)) => 1
  ```

- **cond** passes back multiple values from the last form in the selected clause, but not if the clause is only one long (that is, the returned value is the value of the predicate) since the return is conditional. This rule applies even to the last clause, where the return is not really conditional (the implementation is allowed to pass or not to pass multiple values in this case, and so you should not depend on what it does). **t** should be used as the predicate of the last clause if multiple values are desired, to make it clear to the compiler (and any human readers of the code!) that the return is not conditional.
  Examples:

  ```
  (cond ((numberp 4) (values 1 2))) => 1 and 2
  (cond ((oddp 4) 'foo) ((values 1 2))) => 1 and 2
  ;; Confusion reigns
  ```

- The variants of **cond** such as **if**, **when**, **select**, **zl:selectq**, and **zl:dispatch** pass back multiple values from the last form in the selected clause.
  Examples:

```
(if (numberp 'x) (values 1 2) (values 3 4)) => 3 and 4
(if (numberp 82) (values 1 2) (values 3 4)) => 1 and 2
```

- The number of values returned by **prog** depends on the **return** form used to return from the **prog**. **prog** returns all of the values produced by the subform of **return**. (If a **prog** drops off the end it just returns a single **nil**.)

- **do** behaves like **prog** with respect to **return**. All the values of the last *exit-form* are returned.

- **unwind-protect** passes back multiple values from its protected form. Example:

  ```
  (unwind-protect (values 1 2 3)) => 1 and 2 and 3
  ```

- **catch** passes back multiple values from the last form in its body when it exits normally.

- The obsolete special form **zl:*catch** does not pass back multiple values from the last form in its body, because it is defined to return its own second value to tell you whether the **zl:*catch** form was exited normally or abnormally. This is sometimes inconvenient when you want to propagate back multiple values but you also want to wrap a **zl:*catch** around some forms. Usually people get around this problem by using **catch** or by enclosing the **zl:*catch** in a **prog** and using **return** to pass out the multiple values, **returning** through the **zl:*catch**.

20. Scoping

20.1 Lexical Scoping

Symbolics Common Lisp has a lexically scoped interpreter and compiler. The compiler and interpreter implement the same language.

Consider the following example:

```
(defun fun1 (x)
  (fun2 3 x)
  (fun3 #'(lambda (y) (+ x y)) x 4))
```

This function passes an *internal lambda* to **fun3**. Observe that the internal lambda references the variable **x**, which is neither a lambda variable nor a local variable of this lambda. Rather, it is a variable local to the lambda's *lexical parent*, **fun1**. **fun3** receives as an argument a *lexical closure*, that is, a presentation of the internal lambda in an environment where the variable **x** can be accessed. **x** is a *free lexical variable* of the internal lambda; the closure is said to be a closure of the free lexical variables, specifically in this case, **x**.

Lexical closures, created by reference to internal functions, are to be distinguished from *dynamic closures*, which are created by the **zl:closure** function and the **zl:let-closed** special form. Dynamic closures are closures over *special* variables, while lexical closures are closures over *lexical*, *local* variables. Invocation of a dynamic closure, as a function, causes special variables to be bound. Invocation of a lexical closure simply provides the necessary data linkage for a function to run in the environment in which the closure was made.

Both the compiler and the interpreter support the accessing of lexical variables. The compiler and interpreter also support, in Symbolics Common Lisp as well as Zetalisp, the Common Lisp lexical function and macro definition special forms, **flet**, **labels**, and **macrolet**.

Note that access to lexical variables is true access to the instantiation of the variable and is not limited to the access of values. Thus, assuming that **map-over-list** maps a function over a list in some complex way, the following function works as it appears to, and finds the maximum element of the list.

```
(defun find-max (list)
  (let ((max nil))
    (map-over-list
      #'(lambda (element)
          (when (or (null max)
                    (> element max))
            (setq max element)))
      list)
    max))
```

20.1.1 Lexical Environment Objects and Arguments

Macro-expander functions, the actual functions defined by **defmacro**, **macro**, and **macrolet**, are called with two arguments – *form* and *environment*. Special form implementations used by the interpreter are also passed these two arguments. Macro-expander functions defined by files created prior to the implementation of lexical scoping are passed only a *form* argument, for compatibility.

The *environment* argument allows evaluations and expansions performed by the macro-expander function or the special form interpreter function to be performed in the proper lexical context. The *environment* argument is utilized by the macro-expander function in certain unusual circumstances:

- A macro-expander function explicitly calls **macroexpand** or **macroexpand-1** to expand some code appearing in the form which invoked it. In this case, the environment argument must be passed as a second argument to either of these functions. This is quite uncommon. Most macro-expander functions do not explicitly expand code contained in their calls: **setf** is an example of a macro that does this kind of expansion.

- A macro-expander function explicitly calls **eval** to evaluate, at macro time, an expression appearing in the code which invoked it. In that case, the environment argument must be passed as a second argument to **eval**. This explicit evaluation is even more unusual: almost any use of **eval** by a macro is guaranteed to be wrong, and does not work or do what is intended in certain circumstances. The only known legitimate uses are:

  - A macro determines that some expression is in fact a constant, and computable at macro expand time, and evaluates it. Here, there are no variables involved, so the environment issue is moot.

  - A macro is called with some template code, expressed via backquote, and is expected to produce an instantiation of that template with substitutions performed. Evaluation is the way to instantiate backquoted templates.

The format of lexical environments is an internal artifact of the system. They cannot be constructed or analyzed by user code. It is, however, specified that **nil** represents a null lexical environment.

A macro defined with **defmacro** or **macrolet** can accept its expansion lexical environment (if it needs it for either of the above purposes) as a variable introduced by the lambda-list keyword **&environment** in its argument list.

A macro defined with **macro** receives its lexical environment as its second argument.

20.1.2 Funargs and Lexical Closure Allocation

A *funarg* is a function, usually a lambda, passed as an argument, stored into data structure, or otherwise manipulated as data. Normally, functions are simply called, not manipulated as data. The term funarg is an acronym for *functional argument.* In the following form, two functions are referred to, **zl:sort** and <.

```
(defun number-sort (numbers)
  (sort numbers #'<))
```

zl:sort is being called as a function, but < (more exactly, the function object implementing the < function) is being passed as a funarg.

The major feature of the lexical compiler and interpreter can be described as the support of funargs that reference free lexical variables. Funargs that do not reference free lexical variables also work. For example,

```
(defun data-sort (data)
  (sort data #'(lambda (x y) (< (fun x) (fun y)))))
```

The internal lambda above makes no free lexical references. **zl-user:data-sort** would have worked prior to lexical scoping, and continues to work.

The remainder of this discussion is concerned only with funargs that make free lexical references.

The act of evaluating a form such as

```
#'(lambda (x) (+ x y))
```

produces a lexical closure. (Of course, if we are talking about compiled code, the form is never evaluated. In that case, we are talking about the time in the execution of the compiled function that corresponds to the time that the form would be evaluated.) It is that closure that represents the funarg that is passed around.

Funarg closures can be further classified by usage as *downward funargs* and *upward funargs.* A *downward funarg* is one that does not survive the function call that created the closure. For example:

```
(defun magic-sort (data parameter)
  (sort data #'(lambda (x y) (< (funk x parameter)
                                (funk y parameter)))))
```

In this example, **zl:sort** is passed a lexical closure of the internal lambda. **zl:sort** calls this closure many times to do comparisons. When **zl-user:magic-sort** returns its value, no function or data structure is referencing that closure in any way. That closure is being used as a *downward* funarg; it does not survive the call to **zl-user:magic-sort.**

In this example,

```
(defun make-adder (x)
  #'(lambda (y) (+ x y)))
```

the closure of the internal lambda is returned from the activation of **zl-user:make-adder**, and survives that activation. The closure is being used as an *upward funarg.*

The creation of lexical closures involves the allocation of storage to represent them. This storage can either be allocated on the stack or in the heap. Storage allocated in the heap remains allocated until all references to it are discarded and it is garbage collected. Storage allocated on the stack is transient, and is deallocated when the stack frame in which it is allocated is abandoned. Stack-allocated closures are more efficient, and thus to be desired. Stack-allocated closures can only be used when a funarg is used as a downward funarg. Closures of upward funargs must be allocated in the heap.

Funargs can be passed to any functions. These functions might well store them in permanent data structure, or return them nonlocally, or cause other upward use. Therefore, the compiler and interpreter, in general, must and do assume potential upward use of all funargs. Thus, they cause their closures to be allocated in the heap unless special measures are taken to convey the guarantee of downward-only use. Note that the more general (heap-allocated) closure is guaranteed to work in all cases.

The ephemeral garbage collector substantially reduces the overhead of heap allocation of short-lived objects. Thus, you might be able to ignore these issues entirely, and let the system do as well as it can without additional help.

20.1.2.1 The sys:downward-function And sys:downward-funarg Declarations

There are two ways to convey the guarantee of downward-only use of a funarg. These are the **sys:downward-function** and **sys:downward-funarg** declarations.

20.1.2.2 sys:downward-function Declaration

The declaration **sys:downward-function,** in the body of an internal lambda, guarantees to the system that lexical closures of the lambda in which it appears are only used as downward funargs, and never survive the calls to the procedure that produced them. This allows the system to allocate these closures on the stack.

```
(defun special-search-table (item)
  (block search
    (send *hash-table* :map-hash
          #'(lambda (key object)
              (declare (sys:downward-function))
              (when (magic-function key object item)
                (return-from search object))))))
```

Here, the **:map-hash** message to the hash table calls the closure of the internal lambda many times, but does not store it into permanent variables or data structure, or return it "around" **special-search-table**. Therefore, it is guaranteed that the closure does not survive the call to **special-search-table**. It is thus safe to allow the system to allocate that closure on the stack.

Stack-allocated closures have the same lifetime (*extent*) as **&rest** arguments and lists created by **with-stack-list** and **with-stack-list***, and require the same precautions. See the section "**&rest** Lambda-list Keyword" in *Symbolics Common Lisp: Language Dictionary*.

20.1.2.3 sys:downward-funarg Declaration

The **sys:downward-funarg** declaration (not to be confused with **sys:downward-function**) permits a procedure to declare its intent to use one or more of its arguments in a downward manner. For instance, **zl:sort**'s second argument is a funarg, which is only used in a downward manner, and is declared this way. The second argument to **process-run-function** is a good example of a funarg that is not downward. Here is an example of a function that uses and declares its argument as a downward funarg.

```
(defun search-alist-by-predicate (alist predicate)
  (declare (sys:downward-funarg predicate))
  ;; Traditional "recursive" style, for variety.
  (if (null alist)
      nil
      (let ((element (car list))
            (rest (cdr list))
        (if (funcall predicate (car element))
            (cdr element)
            (search-alist-by-predicate rest predicate)))))
```

This function only calls the funarg passed as the value of **predicate**. It does not store it into permanent structure, return it, or throw it around **search-alist-by-predicate**'s activation.

The reason you so declare the use of an argument is to allow the system to deduce guaranteed downward use of a funarg without need for the **sys:downward-function** declaration. For instance, if **search-alist-by-predicate** were coded as above, we could write

```
(defun look-for-element-in-tolerance (alist required-value tolerance)
  (search-alist-by-predicate alist
    #'(lambda (key)
        (< (abs (- key required-value)) tolerance))))
```

to search the keys of the list for a number within a certain tolerance of a required value. The lexical closure of the internal lambda is automatically allocated by the system on the stack because the system has been told that any funarg used as the first argument to **search-alist-by-predicate** is used only in a downward manner. No declaration in the body of the lambda is required.

All appropriate parameters to system functions have been declared in this way.

There are two possible forms of the **sys:downward-funarg** declaration:

(declare (sys:downward-funarg *var1 var2* **...)**
Declares the named variables, which must be parameters (formal arguments) of the function in which this declaration appears, to have their values used only in a downward fashion. This affects the generation of closures as functional arguments to the function in which this declaration appears: it does not directly affect the function itself. Due to an implementation restriction, *var-i* cannot be a keyword argument.

(declare (sys:downward-funarg *))
Declares guaranteed downward use of all functional arguments to this function. This is to cover closures of functions passed as elements of **&rest** arguments and keyword arguments.

Notes:

The special forms **flet** and **labels** (additions to Zetalisp from Common Lisp) generate lexical closures if necessary. The compiler decides how to allocate a closure generated by **flet** or **labels** after analysis of the use of the function defined by the use of **flet** or **labels**.

It is occasionally appropriate to introduce the **sys:downward-funarg** and **sys:downward-function** (as well as other) declarations into the bodies of functions defined by **flet** and **labels**.

There is no easy way to see if code allocates lexical closures on the heap or on the stack; if disassembly of a compiled function reveals a call to **sys:make-lexical-closure**, heap allocation is indicated.

20.1.3 flet, labels, And macrolet Special Forms

flet *((name args function-body...) ...) body...* *Special Form*

Defines named internal functions. **flet** (function **let**) defines a lexical scope, *body*, in which these names can be used to refer to these functions. *((name args function-body...) ...)* is a list of clauses, each of which defines one function. Each clause of the **flet** is identical to the cdr of a **defun** special form; it is a function name to be defined, followed by an argument list, possibly declarations, and function body forms. **flet** is a mechanism for defining internal subroutines whose names are known only within some local scope.

Functions defined by the clauses of a single **flet** are defined "in parallel", similar to **let**. The names of the functions being defined are not defined and not accessible from the bodies of the functions being defined. The **labels** special form is used to meet those requirements. See the special form **labels**, page 524.

Here is an example of the use of **flet**:

```
(defun triangle-perimeter (p1 p2 p3)
  (flet ((squared (x) (* x x)))
    (flet ((distance (point1 point2)
             (sqrt (+ (squared (- (point-x point1)
                                  (point-x point2)))
                      (squared (- (point-y point1)
                                  (point-y point2)))))))
      (+ (distance p1 p2)
         (distance p2 p3)
         (distance p1 p3)))))
```

flet is used twice here, first to define a subroutine **cl:squared** of **cl:triangle-perimeter**, and then to define another subroutine, **cl:distance**. Note that since **cl:distance** is defined within the scope of the first **flet**, it can use **cl:squared**. **cl:distance** is then called three times in the body of the second **flet**. The names **cl:squared** and **cl:distance** are not meaningful as function names except within the bodies of these **flets**.

Note that functions defined by **flet** are internal, lexical functions of their containing environment. They have the same properties with respect to lexical scoping and references as internal lambdas. They can make free lexical references to variables of that environment and they can be passed as *funargs* to other procedures. Functions defined by **flet**, when passed as funargs, generate closures. The allocation of these closures, that is, whether they appear on the stack or in the heap, is controlled in the same way as for internal lambdas. See the section "Funargs and Lexical Closure Allocation", page 519.

Here is an example of the use, as a funarg, of a closure of a function defined by **flet**.

```
(defun sort-by-closeness-to-goal (list goal)
  (flet ((closer-to-goal (x y)
           (< (abs (- x goal)) (abs (- y goal)))))
    (sort list #'closer-to-goal)))
```

This function sorts a list, where the sort predicate of the (numeric) elements of the list is their absolute distance from the value of the parameter **goal**. That predicate is defined locally by **flet**, and passed to **zl:sort** as a funarg.

Note that **flet** (as well as **labels**) defines the use of a name as a function, not as a variable. Function values are accessed by using a name as the car of a form or by use of the **function** special form (usually expressed by the reader macro **#'**).

Within its lexical scope, **flet** can be used to redefine names that refer to globally defined functions, such as **zl:sort** or **cdar**, though this is not recommended for stylistic reasons. This feature does, however, allow you to bind names with **flet** in an unrestricted fashion, without binding the name of some other function that you might not know about (such as **sys:number-into-array**), and thereby causing other functions to malfunction. This occurs because **flet** always creates a lexical binding, not a dynamic binding. Contrast this with **let**, which usually creates a lexical binding, unless the variable being bound is declared special, in which case it creates a dynamic binding.

flet can also be used to redefine function names defined by enclosing uses of **flet** or **labels**.

labels *((name args function-body...) ...) body...* *Special Form*

Identical to **flet** in structure and purpose, but has slightly different scoping rules. It, too, defines one or more functions whose names are made available within its body. In **labels**, unlike **flet**, however, the functions being defined can refer to each other mutually, and to themselves, recursively. Any of the functions defined by a single use of **labels** can call itself or any other; there is no order dependence. Although **flet** is analogous to **let** in its parallel binding, **labels** is not analogous to **let***.

labels is in all other ways identical to **flet**. It defines internal functions that can be called, re-redefined, passed as funargs, and so on.

Functions defined by **labels**, when passed as funargs, generate closures. The allocation of these closures, that is, whether they appear on the stack or in the heap, is controlled in the same way as for internal lambdas. See the section "Funargs and Lexical Closure Allocation", page 519.

Here is an example of the use of **labels**:

```
(defun combinations (total-things at-a-time)
  ;; This function computes the number of combinations of
  ;; total-things things taken at-a-time at a time.
  ;; There are more efficient ways, but this is illustrative.
  (labels ((factorial (x)
             (permutations x x))
           (permutations (x n)          ;x things n at a time
             (if (= n 1)
                 x
                 (* x (permutations (1- x) (1- n))))))
    (/ (permutations total-things at-a-time)
       (factorial at-a-time))))
```

macrolet *((name args macro-body...) ...) body...* *Special Form*

Defines, within its scope, a macro. It establishes a symbol as a name denoting a macro, and defines the expander function for that macro. **defmacro** does this globally; **macrolet** does it only within the (lexical) scope of its body. A macro so defined can be used as the car of a form within this scope. Such forms are expanded according to the definition supplied when interpreted or compiled.

The syntax of **macrolet** is identical to that of **flet** or **labels**: it consists of clauses defining local, lexical macros, and a body in which the names so defined can be used. *((name args macro-body...) ...) body...* is a list of clauses each of which defines one macro. Each clause is identical to the cdr of a **defmacro** form: it has a name being defined (a symbol), a macro pseudo-argument list, and an expander function body.

The pseudo-argument list is identical to that used by **defmacro**. It is a pattern, and can use appropriate lambda-list keywords for macros, including **&environment**. See the section "Lexical Environment Objects and Arguments", page 518.

The following example of **macrolet** is for demonstration only. If the macro **cl:square** needed to be open-coded, was long and cumbersome, or was used many times, then the use of **macrolet** would be suggested.

```
(defun square-coordinates (point)
  (macrolet ((square (x) `(* ,x ,x)))
    (setf (point-x point) (square (point-x point))
          (point-y point) (square (point-y point)))))


(defstruct point x y) => POINT
(setq p1 (make-point :x 3 :y 4)) => #S(POINT :X 3 :Y 4)
(square-coordinates p1) => 16
```

It is important to realize that macros defined by **macrolet** are run (when the compiler is used), at compile time, not run-time. The expander functions for such macros, that is, the actual code in the body of each **macrolet** clause, cannot attempt to access or set the values of variables of the function containing the use of **macrolet**. Nor can it invoke run-time functions, including local functions defined in the lexical scope of the **macrolet** by use of **flet** or **labels**. The expander function can freely generate code that uses those variables and/or functions, as well as other macros defined in its scope, including itself.

There is an extreme subtlety with respect to expansion-time environments of **macrolet**. It should not affect most uses. The macro-expander functions are closed in the global environment; that is, no variable or function bindings are inherited from any environment. This also means that macros defined by **macrolet** cannot be used in the expander functions of other macros defined by **macrolet** within the scope of the outer **macrolet**. This does not prohibit either of the following:

- Generation of code by the inner macro that refers to the outer one.

- Explicit expansion (by **macroexpand** or **macroexpand-1**), by the inner macro, of code containing calls to the outer macro. Note that explicit environment management must be utilized if this is done. See the section "Lexical Environment Objects and Arguments", page 518.

21. Flow of Control

21.1 Introduction to Flow of Control

Symbolics Common Lisp provides a variety of structures for flow of control.

Function application is the basic method for construction of programs. Operations are written as the application of a function to its arguments. Usually, Lisp programs are written as a large collection of small functions, each of which implements a simple operation. These functions operate by calling one another, and so larger operations are defined in terms of smaller ones.

A function can always call itself in Lisp. The calling of a function by itself is known as *recursion*; it is analogous to mathematical induction.

The performing of an action repeatedly (usually with some changes between repetitions) is called *iteration*, and is provided as a basic control structure in most languages. The *do* statement of PL/I, the *for* statement of ALGOL/60, and so on are examples of iteration primitives. Symbolics Common Lisp provides two general iteration facilities: **do** and **loop**, as well as a variety of special-purpose iteration facilities. (**loop** is sufficiently complex that it is explained in its own section. See the section "The **loop** Iteration Macro", page 537.) There is also a very general construct to allow the traditional "goto" control structure, called **prog**.

A *conditional* construct is one that allows a program to make a decision, and do one thing or another based on some logical condition. Lisp provides the simple one-way conditionals **and** and **or**, the simple two-way conditional **if**, and more general multi-way conditionals such as **cond** and **case**. The choice of which form to use in any particular situation is a matter of personal taste and style.

There are some *nonlocal exit* control structures, analogous to the *leave*, *exit*, and *escape* constructs in many modern languages.

The general ones are **catch** and **throw**; there is also **return** and its variants, used for exiting the iteration constructs **do**, **loop**, and **prog**.

Symbolics Common Lisp also provides a coroutine capability and a multiple-process facility. See the section "Processes" in *Internals, Processes, and Storage Management*. There is also a facility for generic function calling using message passing. See the section "Flavors", page 353.

21.2 Conditionals

21.2.1 Conditional Functions

if *condition true* &rest *false*	The simplest conditional form. Corresponds to the `if-then-else` construct. Returns whatever evaluation of the selected form returns.
cond &rest *clauses*	Selects and evaluates the first clause whose test evaluates to non-**nil**.
cond-every &body *clauses*	Like **cond**, but executes every clause whose predicate is satisfied, not just the first. Returns the value of the last form in the last clause executed.
and &rest *body*	Evaluates each form; returns **nil** if any form evaluates to **nil**. Returns the value of the last form if every form evaluates to non-**nil** values.
or &rest *body*	Evaluates each form until it encounters a form that evaluates to a non-**nil** value; returns the value of that form, or **nil** if all forms evaluate to **nil**.
not *(x)*	**not** returns **t** if *x* is **nil**, else **nil**. **null** is the same as **not**; both functions are included for the sake of clarity. Use **null** to check whether something is **nil**; use **not** to invert the sense of a logical value.
when *condition* &rest *body*	Evaluates the forms in *body* when *condition* returns non-**nil**, and returns the value(s) of the last form evaluated. Returns **nil** when *condition* returns **nil**.
unless *condition* &rest *body*	Evaluates the forms in *body* when *condition* returns **nil**, and returns the value of the last form evaluated. Returns **nil** when *condition* returns a non-**nil** value.
select *test-object* &body *clauses*	Selects one of its clauses for execution by comparing the value of a form against various constants. Returns **nil** or the value of the last clause evaluated. Same as **zl:selectq**, except that the elements of the tests are evaluated before they are used.

selector *test-object test-function* &body *clauses*	The same as **select**, except that instead of using **eq** as the comparison function, **selector** allows the user to specify the test function to use.
selectq-every *object* &body *clauses*	Executes every selected clause, not just the first one. Returns only the value of the last form in the last selected clause.
case *test-object* body *clauses*	Selects one of its clauses for execution by comparing a value to various constants. Allows an explicit **t** clause. Returns **nil**, or the value of the last clause evaluated.
ccase *object* &body *body*	"Continuable exhaustive case." Similar to **case**, but does not allow an explicit **t** clause. Signals a proceedable error if no clause is satisfied; can continue from error by accepting new value from user and restarting tests.
ecase *object* &body *body*	"Exhaustive case," or "error-checking case." Similar to **case**, but does not allow an explicit **t** clause. Signals an error if no clause is satisfied. It is not permissible to continue from this error.
typecase *object* &body *body*	Selects one of its clauses by examining the type of an object. Returns **nil**, or the value of the last clause evaluated. Allows explicit **otherwise** or **t** clause.
ctypecase *object* &body *body*	"Continuable exhaustive type case." Like **typecase**, but does not allow an explicit **otherwise** or **t** clause. Signals a proceedable error if no clause is satisfied.
etypecase *object* &body *body*	"Exhaustive type case," or "error-checking type case." Like **typecase**, but does not allow an explicit **otherwise** or **t** clause. Signals an error if no clause is satisfied. It is not permissible to continue from this error.
zl:selectq *test-object* &body *clauses*	Selects a clause for execution by comparing the value of a form against various constants. Does not evaluate its test elements. Returns **nil**, or

	the value of the last clause evaluated. Accepts **otherwise** or **t**.
zl:caseq *test-object* &body *clauses*	The same as **zl:selectq**, but does not allow **otherwise** or **t** clauses.
zl:typecase *object* &body *body*	Selects forms to evaluate depending on the type of some object. Returns **nil**, or the value of the last clause evaluated. Allows **otherwise** clause.
zl:dispatch *ppss word* &body *clauses*	Selects one of its clauses for execution by comparing the value of a form against various constants. The same as **select** but the *key* is obtained by evaluating `(ldb byte-specifier` *number*). Returns **nil**, or the value of the form evaluated.

Also see **defselect**, a special form for defining a function whose body is like a **zl:selectq**.

21.3 Blocks and Exits

block and **return-from** are the primitive special forms for premature exit from a piece of code. **block** defines a place that can be exited, and **return-from** transfers control to such an exit.

block and **return-from** differ from **catch** and **throw** in their scoping rules. **block** and **return-from** have lexical scope; **catch** and **throw** have dynamic scope.

21.3.1 Blocks and Exits Functions and Variables

block *name* &body *body*	Evaluates each *form* in sequence and normally returns the (possibly multiple) values of the last *form*.
return-from *block-name values*	Exits from a named **block** or a construct like **do** or **prog** that establishes an implicit block around its body. Returns **nil**, zero values, or multiple values depending on the syntax used when invoking the function.

return &optional *values*	Exits from a construct like **do** or an unnamed **prog** that establishes an implicit block around its body. Returns **nil**, zero values, or multiple values depending on the syntax used when invoking the function.
compiler:*return-style-checker-on*	This style-checker variable is attached to **return** and **return-from** and controls the display of compiler messages for invalid formats of these functions.
zl:return-list *form*	Obsolete function; use `(return(values-list` *list*`))` instead.

See the section "Nonlocal Exits", page 533.

21.4 Transfer of Control

tagbody and **go** are the primitive special forms for unstructured transfer of control. **tagbody** defines places that can receive a transfer of control, and **go** transfers control to such a place.

21.4.1 Transfer of Control Functions

go *tag*	Transfers control within a **tagbody** form, or a construct like **do** or **prog** that uses an implicit **tagbody**. *tag* can be a symbol or an integer.
tagbody &body *forms*	Processes each element of the body in sequence, ignoring *tag*(s) and evaluating *statement*(s). If a `(go` *tag*`)` form is evaluated, during evaluation of a statement, **tagbody** transfers control to the innermost *tag* that is equal to the *tag* in the **go** form. A *tag* can be a symbol or an integer.

21.5 Iteration

21.5.1 Iteration Functions

do (*varforms* ...) (*endtest exit-forms* ...) **&body** *body* &whole *form* &rest *ignore* &environment *env*

Provides a generalized iteration facility using "index variables" to control the number of iterations and an end-test to determine when the iteration terminates. The index variable clauses are evaluated in parallel, rather than sequentially. You can optionally specify a return value for **do**.

do* &whole *form* &rest *ignore* &environment *env*

Like **do**, except that the index variable clauses are evaluated sequentially, rather than in parallel.

dolist *(var listform* &optional *resultform)* &body *forms*

Performs *forms* once for each element in the list that is the value of *listform*, with *var* bound to the successive elements. Returns value of *resultform* if the latter is specified.

dotimes *(var countform* &optional *resultform)* &body *forms*

Performs *forms* the number of times given by the value of *countform*, with *var* bound to **0**, **1**, and so forth, on successive iterations. Returns value of *resultform* if the latter is specified.

prog &whole *form* &rest *l* &environment *env*

Provides temporary variables, sequential evaluation of forms and a "goto" facility using tags. The binding of the temporary initialization variables is done in parallel. The **do**, **catch**, and **throw** forms are preferred over **prog**.

prog* &whole *form* &rest *l* &environment *env*

Just like **prog**, except that the binding of the temporary initialization variables is done sequentially.

zl:do-named &whole *form* &rest *ignore* &environment *env*

Works like **do**, but allows a return from a named outer **do** form while you are within an inner **do**. As in **do**, the index variable clauses are evaluated in parallel.

zl:do*-named &whole *form* &rest *ignore* &environment *env*
: Works like **zl:do-named** in allowing a return from a named outer **do** form while within an inner **do**. As in **do***, the index variable clauses are evaluated in sequence.

zl:dolist *(var form)* &body *body*
: Performs *body* once for each element in the list that is the value of *form*, with *var* bound to the successive elements. Similar to **dolist**.

zl:dotimes *(var form)* &body *body*
: Performs *body* the number of times given by the value of *form*, with *var* bound to **0**, **1**, and so on, on successive iterations. Similar to **dotimes**.

zl:keyword-extract *keylist keyvar keyword* &optional *flags* &body *otherwise*
: Obsolete. Use the **&key** lambda-list keyword to create functions that take keyword arguments.

21.6 Nonlocal Exits

catch and **throw** are special forms used for nonlocal exits. **catch** evaluates forms; if a **throw** occurs during the evaluation, **catch** immediately returns (possibly multiple) values specified by **throw**.

catch and **throw** differ from **block** and **tagbody** in their scoping rules. **catch** and **throw** have dynamic scope; **block** and **tagbody** have lexical scope.

For example:

```
(catch 'done
  (ask-database <pattern>
                #'(lambda (x) (when (nice-p x)
                                (throw 'done x)))))
```

See the section "Blocks and Exits", page 530.

zl:*catch and **zl:*throw** are supported for compatibility with Maclisp. **catch** can be used with **zl:*throw**, and **zl:*catch** can be used with **throw**. If control exits normally, the returned values depend on whether **catch** or **zl:*catch** is used. If control exits abnormally, the returned values depend on whether **throw** or **zl:*throw** is used.

The old Maclisp **catch** and **throw** macros are not supported.

21.6.1 Nonlocal Exit Functions

catch *tag* &body *body*

Used with **throw** for nonlocal exits. Evaluates *tag* to obtain an object that is the "tag" of the **catch.** Then, unless **catch** encounters a **throw,** it evaluates *body* forms in sequence, and returns the value(s) of the last form in the *body.*

throw *tag value*

Used with **catch** for nonlocal exits. Evaluates *tag* to obtain an object that is the "tag" of the **throw**. Evaluates *value*; finds the innermost **catch** whose "tag" is **eq** to the "tag" of the **throw.** Causes **catch** to abort the evaluation of its body forms and to return all values that result in evaluating *value.*

unwind-protect *protected-form* &rest *clean-up forms*

Evaluates *protected-form* and when the latter attempts to exit out of the **unwind-protect,** evaluates *clean-up forms.* Returns the value(s) of *protected-form.*

unwind-protect-case (&optional *aborted-p-var) body-form* &rest *cleanup-clauses*

Executes *body-form*; generates cleanup forms from *cleanup-clauses* and executes them depending on the condition specified by the keywords **:normal, :abort, :always.**

sys:without-aborts ([*identifier*] *reason format-args* ...) *body* ...

Encloses code that should not be aborted. Is wrapped around all **unwind-protect** cleanup handlers. Intercepts abort attempts by user (not abort attempts by program), and interacts with the user to postpone or execute the abort attempt. Can be nullified with **sys:with-aborts-enabled.**

sys:with-aborts-enabled (*identifiers* ...) *body* ...

Cancels out one or more invocations of **sys:without-aborts.**

zl:*catch *tag* &body *body*

Obsolete. Use **catch** instead.

zl:*throw *tag value*

Obsolete. Use **throw** instead.

21.7 Mapping

Mapping is a type of iteration in which a function is successively applied to pieces of a list. There are several options for the way in which the pieces of the list are chosen and for what is done with the results returned by the applications of the function.

In general, the mapping functions take two or more arguments. The first argument must be a function, and the second and subsequent arguments must be lists. (Function **map** is a special case, and is discussed elsewhere. See the section "Sequences", page 187.)

For example:

```
(mapcar f x1 x2 ... xn)
```

In this case *f* must be a function of *n* arguments. **mapcar** proceeds down the lists *x1, x2, ..., xn* in parallel. The first argument to *f* comes from *x1*, the second from *x2*, and so on. The iteration stops as soon as any 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 the function **circular-list** in *Symbolics Common Lisp: Language Dictionary*.

Sometimes a **do** or a straightforward recursion is preferable to a **zl:map**; however, the mapping functions should be used wherever they naturally apply because this increases the clarity of the code.

Often *f* is a lambda-expression, rather than a symbol. For example:

```
(mapcar (function (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.

21.7.1 Mapping Functions

map *result-type function* &rest *sequences*	Applies *function* to *sequences*; returns a new sequence such that element *j* of the new sequence is the result of applying *function* to element *j* of each of the argument sequences. *result-type* specifies the type of the result sequence.
mapcar *fcn list* &rest *more-lists*	Applies *fcn* to *list* and to successive elements

	of that list. Accumulates and returns the results of successive calls to *fcn* using **list**.
mapcan *fcn list* &rest *more-lists*	Like **mapcar**, except that it uses **nconc** instead of **list** to accumulate and return its results.
mapc *fcn list* &rest *more-lists*	Like **mapcar**, except that it does not return any useful value.
maplist *fcn list* &rest *more-lists*	Applies *fcn* to *list* and to successive sublists of that list rather than to successive elements. Accumulates and returns the results of successive calls to *fcn* using **list**.
mapcon *fcn list* &rest *more-lists*	Like **maplist**, except that it uses **nconc** instead of **list** to accumulate and return its results.
mapl *fcn list* &rest *more-lists*	Like **maplist**, except that it does not return any useful value.
zl:map *fcn list* &rest *more-lists*	Applies *fcn* to *list* and to successive sublists of that list rather than to successive elements. Does not return any useful value. The same as **mapl**.

Here is a table showing the relations between the six map functions.

		applies function to	
		successive sublists	successive elements
	its own second argument	map	mapc
returns	list of the function results	maplist	mapcar
	nconc of the function results	mapcon	mapcan

There are also functions (**zl:mapatoms** and **zl:mapatoms-all**) for mapping over all symbols in certain packages. See the section "Package Iteration", page 661.

You can also do what the mapping functions do in a different way by using **loop**. See the section "The **loop** Iteration Macro", page 537.

21.8 The loop Iteration Macro

21.8.1 Introduction To loop

The loop macro provides a programmable iteration facility.

The basic structure of a **loop** is as follows:

(**loop** *iteration clauses*
  do
  body) ; loop alone returns **nil**

The *iteration clauses* control the number of times the *body* will be executed. When any iteration clause finishes, the body stops being executed. The word **do** is the keyword which introduces the body of the loop, and as such must be placed between the iteration clauses and the body.

The general approach is that a form introduced by the word **loop** 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 can 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 Symbolics Common Lisp implementation of **loop** is an extension of the Common Lisp specification for this macro in Guy L. Steele's *Common Lisp: the Language.* **loop** works identically in both Symbolics Common Lisp and Zetalisp.

Note that **loop** forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English.

The basic discussion of **loop** deals with the following topics:

- **loop** Clauses
- **loop** Synonyms

The advanced discussion of **loop** deals with the following topics:

- Destructuring
- The Iteration Framework
- Iteration Paths

21.8.2 loop Clauses

Internally, **loop** constructs a **prog** that includes variable bindings, preiteration (initialization) code, postiteration (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).

A *clause* consists of the keyword symbol and any Lisp forms and keywords with which it deals. For example:

```
(loop for x in l
      do (print x))
```

contains two clauses, "`for x in l`" and "`do (print x)`". Certain parts of the clause are described as being *expressions*, such as **(print x)** in the example 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.

loop uses print-name equality to compare keywords so that **loop** forms can be written without package prefixes; in Lisp implementations that do not have packages, **eq** is used for comparison.

Bindings and iteration variable steppings can be performed either sequentially or in parallel, which affects how the stepping of one iteration variable can depend on the value of another. The syntax for distinguishing the two is described with the corresponding clauses. When a set of things is "in parallel", all of the bindings produced are performed in parallel by a single lambda binding. Subsequent bindings are performed inside that binding environment.

The following groups of **loop** clauses are available:

- Iteration-Driving Clauses
- **loop** Initialization Bindings
- Entrance and Exit
- Side Effects
- Accumulating Return Values for **loop**
- End Tests for **loop**
- Aggregated Boolean Tests for **loop**
- **loop** Conditionalization
- Miscellaneous Other Clauses for **loop**

21.8.2.1 Iteration-Driving Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and 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. In order to not produce strange interactions, iteration-driving clauses are required to precede any clauses that produce "body" code: that is, all except those that produce prologue or epilogue code (**initially** and **finally**), bindings (**with**), the **named** clause, and the iteration termination clauses (**while** and **until**).

21.8.2.2 Iteration in Series and in Parallel

Clauses that drive the iteration can be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped 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**. The form this typically takes is something like:

```
(loop ... for x = (f) and for y = init then (g x) ...)
```

which sets **x** to **(f)** on every iteration, and binds **y** to the value of *init* for the first iteration, and on every iteration thereafter sets it 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 would be best written as:

```
(loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example:

```
(loop for sublist on some-list
      and for previous = 'undefined then sublist
         ...)
```

which is equivalent to the **do** construct:

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((null sublist) ...)
  ...)
```

in terms of stepping, would be better written as:

```
(loop for previous = 'undefined then sublist
      for sublist on some-list
         ...)
```

21.8.2.3 Joining Iteration-Driving Clauses with And

When iteration-driving clauses are joined with **and**, if the token following the **and** is not a keyword that introduces an iteration-driving clause, it is assumed to be the same as the keyword that introduced the most recent clause; thus, the above example showing parallel stepping could have been written as:

```
(loop for sublist on some-list
      and previous = 'undefined then sublist
         ...)
```

The order of evaluation in iteration-driving clauses is that those expressions that are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions that are evaluated each time around the loop are evaluated in order in the body.

21.8.2.4 Iterating with Repeat

One common and simple iteration-driving clause is **repeat**:

For example:

```
(defun ex-loop ()
  (loop repeat 4
        for x from 1 to 10
        do
    (princ x)(princ " "))) => EX-LOOP

(ex-loop) => 1 2 3 4
NIL
```

21.8.2.5 Iterating with For and As

All remaining iteration-driving clauses are subdispatches of the keyword **for**, which is synonymous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration-driving clause implicitly supplies an endtest, the value of this iteration variable as the loop is exited (that is, when the epilogue code is run) is undefined. See the section "The Iteration Framework", page 551.

21.8.2.6 Iteration Keywords and For Clauses

Here are the iteration keywords and all the varieties of **for** clauses. Optional parts are enclosed in curly brackets. The optional argument, *data-type*, is reserved for data type declarations. It is currently ignored.

The optional **by** phrase specifies the size of the step by which to increment or decrement the expression serving as the loop counter. Default step is 1.

repeat *expression* Causes the **loop** to iterate that many times. *Expression* is expected to evaluate to an integer.

for *var* {*data-type*} **from** *expr1* {**to** *expr2*} {**by** *expr3*}
Iterates *upward.* Performs numeric iteration. *var* is initialized to *expr1,* and on each succeeding iteration is incremented by *expr3.* If the **to** phrase is given, the iteration terminates when *var* becomes greater than *expr2.*

for *var* {*data-type*} **from** *expr1* **downto** *expr2* {**by** *expr3*}
Iterates *downward.* Performs numeric iteration. *var* is initialized to *expr1,* and on each succeeding iteration is decremented by *expr3.* The iteration terminates when *var* becomes less then *expr2.*

for *var* {*data-type*} **from** *expr1* {**below** *expr2*} {**by** *expr3*}
Iteration terminates when the variable of iteration, *expr1,* is *greater than or equal* to some terminal value, *expr2.*

for *var* {*data-type*} **from** *expr1* {**above** *expr2*} {**by** *expr3*}
Iteration terminates when the variable of iteration is *less than* or *equal* to some terminal value.

for *var* {*data-type*} **downfrom** *expr1* {**by** *expr2*}
Used to iterate *downward* with no limit.

for *var* {*data-type*} **upfrom** *expr1* {**by** *expr2*}
Used to iterate *upward* with no limit.

for *var* {*data-type*} **in** *expr1* {**by** *expr2*}
Iterates over each of the elements in the list, *expr1*, (its **car**).

for *var* **on** *expr1* {**by** *expr2*}
Iterates over successive sublists of the list, *expr1*, (its **cdr**).

for *var* {*data-type*} = *expr*
On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* {*data-type*} = *form-1* **then** *form-2*
var is bound to *expr1* when the loop is entered, and set to *expr2* (reevaluated) at all but the first iteration.

for *var* {*data-type*} **first** *form-1* **then** *form-2*
Sets *var* to *expr1* on the first iteration, and to *expr2* (reevaluated) on each succeeding iteration.

for *var* {*data-type*} **being** *expr* **and its** *path* **...**

for *var* {*data-type*} **being** {**each|the**} *path* **...**
Provide a user-definable iteration facility. *path* names the manner in which the iteration is to be performed. The ellipsis indicates where various path-dependent preposition/expression pairs can appear. See the section "Iteration Paths", page 553.

21.8.2.7 loop Initialization Bindings

To declare local variables and constants in a loop, use the keyword **with**.

For example:

```
(defun ex-loop-1 ()
  (loop for x from 0 to 4
        with (one four)
        with three = "three"
        doing
    (princ x)(princ " ")
             (setq four x)(setq one "one")
        finally (return (values  one three four)))) => EX-LOOP-1

(ex-loop-1)  => 0 1 2 3 4 one and three and 4
```

21.8.2.8 Keywords for Loop Initialization Bindings

with *var* {*data-type*} {= *expr1*} {**and** *var2* { *data-type*} { = *expr2*}}...

The **with** keyword can be used to establish initial bindings, that is, variables that are local to the loop but are only set once, rather than on each iteration. The optional argument, *data-type*, is currently ignored.

nodeclare *variable-list*

The variables in *variable-list* are noted by **loop** as not requiring local type declarations. This is for compatibility with other implementations of **loop**. Symbolics Common Lisp never uses type declarations.

21.8.2.9 Entrance and Exit

To introduce initialization *(prologue)* code in **loop** use the keyword **initially.**

For example:

```
(loop for x from 1 to 4
      initially (princ "lets count... ")
      doing (princ x)) => lets count... 1234
NIL
```

To introduce exit *(epilogue)* code in **loop** use the keyword **finally.**

For example:

```
(loop for x from 1 to 4
      finally (princ "lets count... ")
      doing (princ x)) => 1234lets count...
NIL
```

21.8.2.10 Entrance and Exit Keywords

initially *expression*

The **initially** keyword introduces *preiteration* or *entrance* code. The *expression* following **initially** is evaluated only once, *after* all initial bindings are made, but *before* the first iteration.

finally *expression* The **finally** keyword introduces *postiteration* or *exit* code. The form following **finally** is evaluated only once, after the loop has terminated for some reason, but before the loop returns. It is not run when the loop is exited with the **return** special form or the **return loop** keyword.

21.8.2.11 Side Effects

The word **do** is the keyword which introduces the body of the loop, and as such must be placed between the iteration clauses and the body.

For example:

```
(loop for x from 1 to 4
      do
  (princ x)) => 1234
NIL
```

do[ing] *expression*

expression is evaluated each time through the loop. **do** and **doing** are equivalent keywords.

21.8.2.12 Accumulating Return Values For loop

The following clauses accumulate a return value for the iteration in some manner. The general form is:

type-of-collection expr {*data-type*} {**into** *var*}

where *type-of-collection* is a **loop** keyword, and *expr* is the thing being "accumulated" somehow. If no **into** 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) has been set to the accumulated result and can 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 reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached.

For 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)))
```

has the same effect as:

```
(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))
  (setq foo-list (cons (foo x) foo-list))
  (setq bar-list (cons (bar x) bar-list))
  (setq baz-list (cons (baz x) baz-list)))
```

except that **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.

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. **collect**, **nconc**, and **append** can all 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 (quotient sum-var count-var)))
```

21.8.2.13 Keywords for Accumulating Return Values For loop

Where present in a keyword name, the square brackets indicate an equivalent form of the keyword. For example, you can use **collect** or **collecting, nconc** or **nconcing,** and so on.

collect[ing] *expr* {**into** *result*}	Causes the values of *expr* on each iteration to be collected into a list, *result.*
nconc[ing] *expr*	Causes the values of *expr* on each iteration to be concatenated together.
append[ing] *expr* {**into** *var*}	Causes the values of *expr* on each iteration to be appended together.

count[ing] *expr* {**into** *var*} {*data-type*}	If *expr* evaluates non-nil, a counter is incremented.
sum[ming] *expr* {*data-type*} {**into** *var*}	Evaluates *expr* on each iteration, and accumulates the sum of all the values.
maximize *expr* {*data-type*} {**into** *var*}	Computes the maximum of *expr* over all iterations.
minimize *expr* {*data-type*} {**into** *var*}	Computes the minimum of *expr* over all iterations.

21.8.2.14 End Tests For loop

The following clauses can be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (for example, from **collect**).

until might be needed, for example, to step through a strange data structure, as in:

```
(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 can also be of use in terminating the iteration.

21.8.2.15 End Test Keywords For loop

while *expr*	If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any), and returning any accumulated value.
until *expr*	If *expr* evaluates to **t**, the loop is exited, performing exit code (if any), and returning any accumulated value.
loop-finish	Causes the iteration to terminate "normally"; epilogue code (if any) is run.

21.8.2.16 Aggregated Boolean Tests For loop

All of these clauses perform some test, and can immediately terminate the iteration depending on the result of that test.

For example:

```
(defun example-using-always (my-list)
  (loop for x in my-list
        always (numberp x)
        finally (return "made it"))) => EXAMPLE-USING-ALWAYS

(example-using-always '(1 2 3)) => "made it"
(example-using-always '(a b c)) => NIL
```

always *expr*	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.
never *expr*	Causes the loop to return **t** if *expr* **never** evaluates non-null.
thereis *expr*	If *expr* evaluates non-**null**, the iteration is terminated and that value is returned, without running the epilogue code. If the loop terminates before *expr* is ever evaluated, the epilogue code is run and the loop returns nil.

21.8.2.17 loop Conditionalization

The keywords **when**, **if-then-else**, and **unless** can be used to "conditionalize" the following clause. Conditionalization clauses can precede any of the side-effecting or value-producing clauses, such as **do**, **collect**, **always**, or **return**.

Multiple conditionalization clauses can appear in sequence. If one test fails, then any following tests in the immediate sequence, and the clause being conditionalized, are skipped.

The format of a conditionalized clause is typically something like:

when *expr1 keyword expr2*

For example:

keyword can be a keyword introducing a side-effecting or value-producing clause.

If *expr2* is the keyword **it**, a variable is generated to hold the value of *expr1* and that variable is substituted for *expr2*. See the section "Conditionalizing with the Keyword **it**", page 549.

21.8.2.18 Conditionalizing Multiple Clauses with And

Multiple clauses can be conditionalized under the same test by joining them with **and**, as in:

```
(loop for i from a to b
      when (zerop (remainder i 3))
        collect i and do (print i))
```

which returns a list of all multiples of **3** from **a** to **b** (inclusive) and prints them as they are being collected.

21.8.2.19 Conditionalizing With If-then-else

If-then-else conditionals can be written using the **else** keyword, as in:

```
(loop for i from 1 to 9
      if (oddp i)
        collect i into odd-numbers
      else collect i into even-numbers
      finally (return even-numbers)) => (2 4 6 8)
```

Multiple clauses can appear in an **else**-phrase, using **and** to join them in the same way as above.

21.8.2.20 Nesting Conditionals

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)
```

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**. Here is a complicated example.

```
(loop for x in l
      when (atom x)
        when (memq x *distinguished-symbols*)
          do (process1 x)
      else do (process2 x)
      else when (memq (car x) *special-prefixes*)
             collect (process3 (car x) (cdr x))
             and do (memorize x)
      else do (process4 x))
```

21.8.2.21 Conditionalizing The return Clause

Useful with the conditionalization clauses is the **return** clause, which causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is:

when *expr1* **return** *expr2*

is equivalent to:

when *expr1* **do (return** *expr2***)**

21.8.2.22 Conditionalizing an Aggregated Boolean Value Clause

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause the iteration to terminate early not to be performed unless the condition succeeds. For example:

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

does not make the **extra-special-significant-p** check unless the **significant-p** check succeeds.

21.8.2.23 Conditionalizing with the Keyword it

In the typical format of a conditionalized clause such as

when *expr1 keyword expr2*

expr2 can be the keyword **it**. If that is the case, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition:

when *expr* **return it**

is equivalent to the clause:

thereis *expr*

and one can collect all non-**null** values in an iteration by saying:

when *expression* **collect it**

If multiple clauses are joined with **and**, the **it** keyword can only be used 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.

21.8.2.24 loop Conditionalizing Keywords

when *expr*	If *expr* evaluates to **nil**, the following clause is skipped, otherwise not.
if *expr* {**else** *expr* **else** *expr* **...**}	If *expr* evaluates to **nil**, the following clause is skipped, otherwise not.
unless *expr* {**else** *expr* **else** *expr* **...**}	If *expr* evaluates to **t**, the following clause is skipped, otherwise not. This is equivalent to `(when (not` *expr*`))`

21.8.2.25 Miscellaneous Other Clauses For loop

named *name*	Gives the **prog** that **loop** generates a name of *name*, so that you can use the **return-from** form to return explicitly out of that particular **loop**.
return *expression*	Immediately returns the value of *expression* as the value of the loop, without running the epilogue code.

21.8.3 loop Synonyms

This facility exists primarily for diehard users of a predecessor of **loop**. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

The **define-loop-macro** macro can be used to make its argument, *keyword*, a **loop** keyword (such as **for**) into a Lisp macro that can introduce a **loop** form.

21.8.4 Destructuring

Destructuring provides you with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example:

```
(loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding **foo** to **a** and **bar** to **(b c)**.

Here's how this might work:

```
(defun ex-destructuring ()
  (loop for x from 1 to 4
        with (one . rest) = '(1 2 3)
        do
    (princ x)(princ " ")
        finally (print one)(print rest)))  => EX-DESTRUCTURING

(ex-destructuring) => 1 2 3 4
1
(2 3)
NIL
```

Consider map-over-properties, in the next example:

```
(defun map-over-properties (fn symbol)
  (loop for (propname propval) on (plist symbol) by 'cddr
        do (funcall fn symbol propname propval)))
```

maps *fn* over the properties on *symbol*, giving it arguments of the symbol, the property name, and the value of that property.

21.8.5 The Iteration Framework

This section describes the way **loop** constructs iterations. It is necessary if you are writing your own iteration paths, and can be useful in clarifying what **loop** does with its input.

loop considers the act of *stepping* to have four possible parts. Each iteration-driving clause has some or all of these four parts, which are executed in this order:

pre-step-endtest
This is an endtest that determines if it is safe to step to the next value of the iteration variable.

steps Variables that get "stepped". This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest
Sometimes you cannot see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps
Other things that need to be stepped. This is typically used for internal

variables that are more conveniently stepped here, or to set up iteration variables that are functions of some internal variable(s) that are actually driving the iteration. This is a list like *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient in just about all iteration-driving clauses that **loop** handles. What is missing is that in most cases, the stepping and testing for the first time through the loop is different from that of all other times. So, what **loop** deals with is two four-tuples as above; one for the first iteration, and one for the rest. The first can be thought of as describing code that immediately precedes the loop in the **prog**, and the second as following the body code – in fact, **loop** does just this, but severely perturbs it in order to reduce code duplication. Two lists of forms are constructed in parallel: one is the first-iteration endtests and steps, the other the remaining-iterations endtests and steps. These lists have dummy entries in them so that identical expressions appear in the same position in both. When **loop** is done parsing all of the clauses, these lists get merged back together such that corresponding identical expressions in both lists are not duplicated unless they are "simple" and it is worth doing.

Thus, one *might* get some duplicated code if one has multiple iterations. Alternatively, **loop** might decide to use and test a flag variable that indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

Note also that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration can be either before or after the stepping. A notable case of this is:

```
(loop for i from 1 to 3 and dummy = (print 'foo)
      collect i) =>
FOO
FOO
FOO
FOO (1 2 3)
```

which prints **foo** *four* times. Certain other constructs, such as **for** *var* **on**, might or might not do this depending on the particular construction.

This problem also means that it might not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration-driving clause implicitly supplies an endtest, then you cannot know the state of the iteration variable when the loop terminates. Although you can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *might* be wrong. Thus:

```
(loop for subl on expr
          ...
      finally (f subl))
```

is incorrect, but:

```
(loop as frob = expr while (g frob)
          ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

21.8.6 Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-driving clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of **loop**. The typical form of an iteration path is

`for` *var* {*data-type*} `being` {`each|the`} *path-name* {*preposition1 expr1*}...

path-name is an atomic symbol that is defined as a **loop** path function. The usage and defaulting of *data-type* is up to the path function. Any number of preposition/expression pairs can be present; the prepositions allowable for any particular path are defined by that path. For example:

```
(loop for x being the array-elements of my-array from 1 to 10
          ...)
```

To enhance readability, *path-name* arguments are usually defined in both the singular and plural forms; this particular example could have been written as:

```
(loop for x being each array-element of my-array from 1 to 10
          ...)
```

Another format, which is not so generally applicable, is:

`for` *var* {*data-type*} `being` *expr0* `and its` *path-name* {*preposition1 expr1*}...

In this format, *var* takes on the value of *expr0* the first time through the loop. Support for this format is usually limited to paths that step through some data structure, such as the "superiors" of something. Thus, we can hypothesize the **cdrs** path, such that:

```
(loop for x being the cdr of '(a b c . d) collect x)
```

but:

```
(loop for x being '(a b c . d) and its cdrs collect x)
 => ((a b c . d) (b c . d) (c . d) d)
```

To satisfy the anthropomorphic among you, **his**, **her**, or **their** can be substituted for the **its** keyword, as can **each**. Egocentricity is not condoned. See the section "Predefined Iteration Paths", page 555. Some example uses of iteration paths are shown in that section.

Very often, iteration paths step internal variables that you do not specify, such as an index into some data structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. **loop** provides an additional syntax with which you can provide a variable name to be used as an "internal" variable by an iteration path, with the **using** "prepositional phrase".

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) (sequence s)
      ...)
```

which says that the variable **i** should be used to hold the index of the array being stepped through, and the variable **s** should be bound to the array. The particular keywords that can be used are defined by the iteration path; the **index** and **sequence** keywords are recognized by all **loop** sequence paths. See the section "Sequence Iteration", page 556. Note that any individual **using** phrase applies to only one path; it is parsed along with the "prepositional phrases". It is an error if the path does not call for a variable using that keyword.

Examples:

```
(setq a (make-array 4)) => #(NIL NIL NIL NIL)
(loop for x being the array-elements of a using (index i) (sequence s)
      doing
  (princ x) (princ " ") (princ i)(princ " ")
      finally (print s)) => NIL 0 NIL 1 NIL 2 NIL 3
#(NIL NIL NIL NIL)
NIL
```

By special dispensation, if a *path-name* is not recognized, then the **default-loop-path** path is invoked upon a syntactic transformation of the original input. Essentially, the **loop** fragment:

```
for var being frob
```

is taken as if it were:

```
    for var being default-loop-path in frob
```

and:

```
    for var being expr and its frob ...
```

is taken as if it were:

```
    for var being expr and its default-loop-path in frob
```

Thus, this "undefined pathname hook" only works if the **default-loop-path** path is defined. Obviously, the use of this "hook" is competitive, since 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 use by large systems that wish to use a special syntax for some feature they provide.

21.8.6.1 loop Iteration Over Hash Tables or Heaps

loop has iteration paths that support iterating over each entry in a hash table or a heap.

```
    (loop for x being the hash-elements of new-coms ...)
    (loop for x being the hash-elements of new-coms with-key k ...)

    (loop for x being the heap-elements of priority-queue ...)
    (loop for x being the heap-elements of priority-queue with-key k ...)
```

This allows *x* to take on the values of successive entries of hash tables or heaps. The body of the loop runs once for each entry of the hash table or heap. For heaps, *x* could have the same value more than once, since the key is not necessarily unique. When looping over hash tables or heaps, the ordering of the elements is undefined.

The **with-key** phrase is optional. It provides for the variable *k* to have the hash or heap key for the particular entry value *x* that you are examining.

The **heap-elements loop** iteration path returns the items in random order and does not provide for locking the heap.

21.8.6.2 Predefined Iteration Paths

loop comes with two predefined iteration path functions; one implements a **mapatoms**-like iteration path facility, and the other is used for defining iteration paths for stepping through sequences.

21.8.6.3 The Interned-symbols Path

The **interned-symbols** iteration path is like a **mapatoms** for **loop**.

```
(loop for sym being interned-symbols ...)
```

iterates over all of the symbols in the current package and its superiors (or, in Maclisp, the current obarray). This is the same set of symbols that **mapatoms** iterates over, although not necessarily in the same order. The particular package to look in can be specified as in:

```
(loop for sym being the interned-symbols in package ...)
```

which is like giving a second argument to **mapatoms**.

In Lisp implementations such as Symbolics Common Lisp with some sort of hierarchical package structure, you can restrict the iteration to be over just the package specified and not its superiors, by using the **local-interned-symbols** path:

```
(loop for sym being the local-interned-symbols  {in package}
          ...)
```

Example:

```
(defun my-apropos (sub-string &optional (pkg package))
  (loop for x being the interned-symbols in pkg
        when (string-search sub-string x)
          when (or (boundp x) (fboundp x) (si:plist x))
            do (print x)))  ; try writing print-interesting-info
 => MY-APROPOS
(my-apropos 'car 'cl-user)  =>
CAR
MAPCAR
NIL
```

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 to the implementation of packages.

21.8.6.4 Sequence Iteration

One very common form of iteration is that over the elements of some object that is accessible by means of an integer index. **loop** defines an iteration path function for doing this in a general way, and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

The Symbolics Common Lisp implementation of **loop** utilizes the Symbolics Common Lisp array manipulation primitives to define both **array-element** and **array-elements** as iteration paths:

```
(define-loop-sequence-path (array-element array-elements)
    aref array-active-length)
```

Then, the **loop** clause:

```
for var being the array-elements of array
```

steps *var* over the elements of *array*, starting from **0**. 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 the same keywords that are accepted by the **loop** arithmetic stepper (**for** *var* **from ...**); they are **by, to, downto, from, downfrom, below,** and **above**, and are interpreted in the same manner. Thus:

```
(loop for var being the array-elements of array
         from 1 by 2
         ...)
```

steps *var* over all of the odd elements of *array*, and:

```
(loop for var being the array-elements of array
         downto 0
         ...)
```

steps in "reverse" order.

```
(define-loop-sequence-path (vector-elements vector-element)
                           vref vector-length notype notype)
```

is how the **vector-elements** iteration path can be defined in NIL (which it is). You can then do such things as:

```
(defun cons-a-lot (item &rest other-items)
  (and other-items
       (loop for x being the vector-elements of other-items
             collect (cons item x))))
```

All such sequence iteration paths allow you to specify the variable to be used as the index variable, by use of the **index** keyword with the **using** prepositional phrase. You can also use the **sequence** keyword with the **using** prepositional phrase to specify the variable to be bound to the sequence.

See the section "Iteration Paths", page 553.

21.8.6.5 Sequence Iteration Macro

define-loop-sequence-path *path-name-or-names fetchfun sizefun &optional sequence-type element-type*

Defines an iteration over the elements of some object that is accessible by means of an integer index.

21.8.6.6 Defining Iteration Paths

A **loop** iteration clause (for example, a **for** or **as** clause) produces, in addition to the code that defines the iteration, variables that must be bound, and preiteration (*prologue*) code. See the section "The Iteration Framework", page 551. This breakdown allows a user interface to **loop** that does not have to 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 returns those items. A function to generate code for a path can be declared to **loop** with the **define-loop-path** function:

The handler is the *path-function* for **define-loop-path**. 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	A list with entries of the form *(preposition expression)*, in the order in which they were collected. This can also include some 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?	**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*), **nil** otherwise. When **t**, *expr* appears in *prepositional-phrases* with the **of** preposition; for example, **(for x being foo and its cdrs)** gets *prepositional-phrases* of **((of foo))**.
allowed-prepositions	The list of allowable prepositions declared for the pathname that caused the path function to be invoked. It and *data* can be used by the path function such that a single function can handle similar paths.
data	The list of "data" declared for the pathname that caused the path function to be invoked. It might, 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 might be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings
: A list of variables that need to be bound. The entries in it can 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 gets bound. All of these variables are bound in parallel; if initialization of one depends on others, it should be done with a **setq** in the *prologue-forms*. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms
: A list of forms that should be included in the **loop** prologue.

the four items of the iteration specification
: The four items: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*. See the section "The Iteration Framework", page 551.

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 next three routines 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. In certain implementations these functions might be implemented as macros.

si:loop-tequal *token keyword*
: The **loop** token comparison function.

si:loop-tmember *token keyword-list*
: The **member** variant of **si:loop-tequal**.

si:loop-tassoc *token keyword-alist*
: The **assoc** variant of **si:loop-tequal**.

si:loop-named-variable *keyword*
: Used when an iteration path function desires to make an internal variable accessible to the user; use instead of **gensym**. Should only be called from within an iteration path function.

define-loop-path *pathname-or-names path-function list-of-allowable-prepositions datum-1 datum-2 ...*
: This 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. The arguments *datum-1*, *datum-2* are optional.

21.8.6.7 An Example Path Definition

Here is an example function that defines the **string-characters** iteration path. This path steps a variable through all of the characters of a string. It accepts the format:

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by:

```
(define-loop-path string-characters string-chars-path
  (of)) => NIL
```

Here is the 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)))
  allowed-prepositions data ; unused variables
  ; 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:
  (cond ((null data-type) (setq data-type 'character)))
  ; We support exactly one "preposition", which is
  ; required, so this check suffices:
  (cond ((null prep-phrases)
         (error  "OF missing in ~S iteration path of ~S"
                 path-name variable)))
  ; We do not support "inclusive" iteration:
  (cond ((not (null inclusive?))
         (zl:ferror nil
           "Inclusive stepping not supported in ~S path ~
            of ~S (prep phrases = ~:S)"
           path-name variable prep-phrases)))
  ; Set up the bindings
  (setq 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
  (setq prologue (list `(setq ,size-var (string-length
                                          ,string-var))))
  ; and return the appropriate items, explained below.
  (list bindings
        prologue
        `(= ,index-var ,size-var)
        nil
        nil
        (list variable `(aref ,string-var ,index-var)
              index-var `(1+ ,index-var)))) => STRING-CHARS-PATH
```

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 (being **gensym**med), and the stepping of it (**1+**) is such that it can be performed safely without an endtest. 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 is advantageous from the standpoint of the optimizations **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 results in some efficiency gains.

If you wanted the index variable in the above definition to be user-accessible through the **using** phrase feature with the **index** keyword, the function would need to be changed in two ways. First, **index-var** should be bound to **(si:loop-named-variable 'index)** instead of **(gensym)**. Secondly, 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:

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(char-n ,string-var ,index-var))
      nil
      nil
      '(= ,index-var ,size-var)
      (list variable '(char-n ,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.

Example:

```
(loop for x being the string-characters of "ABCDEFG"
      doing
  (print (ascii-code x))) =>
65
66
67
68
69
70
71
NIL

(loop for x being the string-characters "abc"
      doing
  (print x))
 => Error: OF missing in STRING-CHARACTERS iteration path of X
```

22. Conditions

22.1 Introduction to Signalling and Handling Conditions

This documentation is tailored for applications programmers. It contains descriptions of all conditions that are signalled by Genera. With this information, you can write your own handlers for events detected by the system or define and handle classes of events appropriate for your own application.

The documentation describes the following major topics.

- Mechanisms for handling conditions that have been signalled by system or application code.
- Mechanisms for defining new conditions.
- Mechanisms that are appropriate for application programs to use to signal conditions.
- All of the conditions that are defined by and used in the system software.

22.1.1 Overview and Definitions of Signalling and Handling

An *event* is "something that happens" during execution of a program. That is, it is some circumstance that the system can detect, like the effect of dividing by zero. Some events are errors – which means something happened that was not part of the contract of a given function – and some are not. In either case, a program can report that the event has occurred, and it can find and execute user-supplied code as a result.

The reporting process is called *signalling*, and subsequent processing is called *handling*. A *handler* is a piece of user-supplied code that assumes control when it is invoked as a result of signalling. Genera includes default mechanisms to handle a standard set of events automatically.

The mechanism for reporting the occurrence of an event relies on flavors. Each standard class of events has a corresponding flavor called a *condition*. For example, occurrences of the event "dividing by zero" correspond to the condition **sys:divide-by-zero**.

The mechanism for reporting the occurrence of an event is called *signalling a condition*. The signalling mechanism creates a *condition object* of the flavor appropriate for the event. The condition object is an instance of that flavor. The instance contains information about the event, such as a textual message to report, and various parameters of the condition. For example, when a program divides a number by zero, the signalling mechanism creates an instance of the flavor **sys:divide-by-zero**.

Handlers are pieces of user or system code that are bound for a particular condition or set of conditions. When an event occurs, the signalling mechanism searches all of the currently bound handlers to find the one that corresponds to the condition. The handler can then access the instance variables of the condition object to learn more about the condition and hence about the event.

Handlers have dynamic scope, so that the handler that is invoked for a condition is the one that was bound most recently.

The condition system provides flexible mechanisms for determining what to do after a handler runs. The handler can try to *proceed*, which means that the program might be able to continue execution past the point at which the condition was signalled, possibly after correcting the error. Any program can designate *restart* points. This facility allows a user to retry an operation from some earlier point in a program.

Some conditions are very specific to a particular set of error circumstances and others are more general. For example, **fs:delete-failure** is a specialization of **fs:file-operation-failure** which is in turn a specialization of **fs:file-error**. You choose the level of condition that is appropriate to handle according to the needs of the particular application. Thus, a handler can correspond to a single condition or to a predefined class of conditions. This capability is provided by the flavor inheritance mechanism.

22.2 How Applications Programs Treat Conditions

This section provides an overview of how applications programs treat conditions.

- A program signals a condition when it wants to report an occurrence of an event.

- A program binds a handler when it wants to gain control when an event occurs.

When the system or a user function detects an error, it signals an appropriate condition and some handler bound for that condition then deals with it.

Conditions are flavors. Each condition is named by a symbol that is the name of a flavor, for example, **sys:unbound-variable, sys:divide-by-zero, fs:file-not-found**. As part of signalling a condition, the program creates a condition object of the appropriate flavor. The condition object contains information about the event, such as a textual message to report and various parameters. For example, a condition object of flavor **fs:file-not-found** contains the pathname that the file system failed to find.

Handlers are bound with dynamic scope, so the most recently bound handler for the condition is invoked. When an event occurs, the signalling mechanism searches all of the current handlers, starting with the innermost handler, for one that can handle the condition that has been signalled. When an appropriate handler is found, it can access the condition object to learn more about the error.

22.2.1 Example of a Handler

condition-case is a simple form for binding a handler. For example:

```
(condition-case ()
   (/ a b)
  (sys:divide-by-zero *infinity*))
```

This form does two things.

- Evaluates **(/ a b)** and returns the result.

- Binds a handler for the **sys:divide-by-zero** condition which applies during the evaluation of **(/ a b)**.

In this example, it is a simple handler that just returns a value. If division by zero happened in the course of evaluating **(/ a b)**, the form would return the value of ***infinity*** instead. If any other error occurred, it would be handled by the system's default handler for that condition or by some other user handler of higher scope.

You can also bind a handler for a predefined class of conditions. For example, the symbol **fs:file-operation-failure** refers to the set of all error conditions in file system operations, such as "file not found" or "directory not found" or "link to nonexistent file", but not to such errors as "network connection closed" or "invalid arguments to **open**", which are members of different classes.

22.2.2 Signalling

You can signal a condition by calling either **signal** or **error**. **signal** is the most general signalling function; it can signal any condition. It allows either a handler or the user to proceed from the error. **error** is a more restrictive version that accepts only error conditions and does not allow proceeding. **error** is guaranteed never to return to its caller.

Both **signal** and **error** have the same calling sequence. The first argument is a symbol that names a condition; the rest are keyword arguments that let you provide extra information about the error. See the section "Signalling Conditions", page 578. Full details on using the signalling mechanism are in that section.

Applications programs rarely need to signal system conditions although they can. Usually, programs which want to signal conditions define their own condition flavor to signal.

Two simpler signalling functions, called **zl:ferror** and **zl:fsignal**, are applicable when you want to signal without defining a new condition. **Note:** These two functions are now obsolete. In new programs, use **error** instead of **zl:ferror**, and **cerror** instead of **zl:fsignal**.

It is very important to understand that signalling a condition is not just the same thing as throwing to a tag. **throw** is a simple control-structure mechanism allowing control to escape from an inner form to an outer form. Signalling is a convention for finding and executing a piece of user-supplied code when one of a class of events occurs. A condition handler might in fact do a **throw**, but it is under no obligation to do so. User programs can continue to use **throw**; it is simply a different capability with a different application.

22.2.3 Condition Flavors

Symbols for conditions are the names of flavors; sets of conditions are defined by the flavor inheritance mechanism. For example, the flavor **lmfs:lmfs-file-not-found** is built on the flavor **fs:file-not-found**, which is built on **fs:file-operation-failure**, which is in turn built on the flavor **error**.

The flavor inheritance mechanism controls which handler is invoked. For example, when a Genera file system operation fails to find a file, it could signal **lmfs:lmfs-file-not-found**. The signalling mechanism invokes the first appropriate handler that it finds, in this case, a handler for **fs:file-not-found**, one for **fs:file-operation-failure**, or one for **error**. In general, if a handler is bound for flavor **a**, and a condition object **c** of flavor **b** is signalled, then the handler is invoked if **(typep c 'a)** is true; that is, if **a** is one of the flavors that **b** is built on.

The symbol **condition** refers to all conditions, including simple, error, and debugger conditions. The symbol **error** refers to the set of all error conditions. Figure 1 shows an overview of the flavor hierarchy.

error is a base flavor for many conditions, but not all. *Simple conditions* are those built on **condition**; *debugger conditions* are those built on **dbg:debugger-condition**. *Error conditions* or *errors* are those built on **error**. For your own condition definitions, whether you decide to treat something as an error or as a simple condition is up to the semantics of the application.

From a more technical viewpoint, the distinction between simple conditions and debugger conditions hinges on what action occurs when the program does not provide its own handler. For a debugger condition, the system invokes the Debugger; for a simple condition, **signal** simply returns **nil** to the caller.

A warning mechanism also exists; the function **warn** is like the function **signal**,

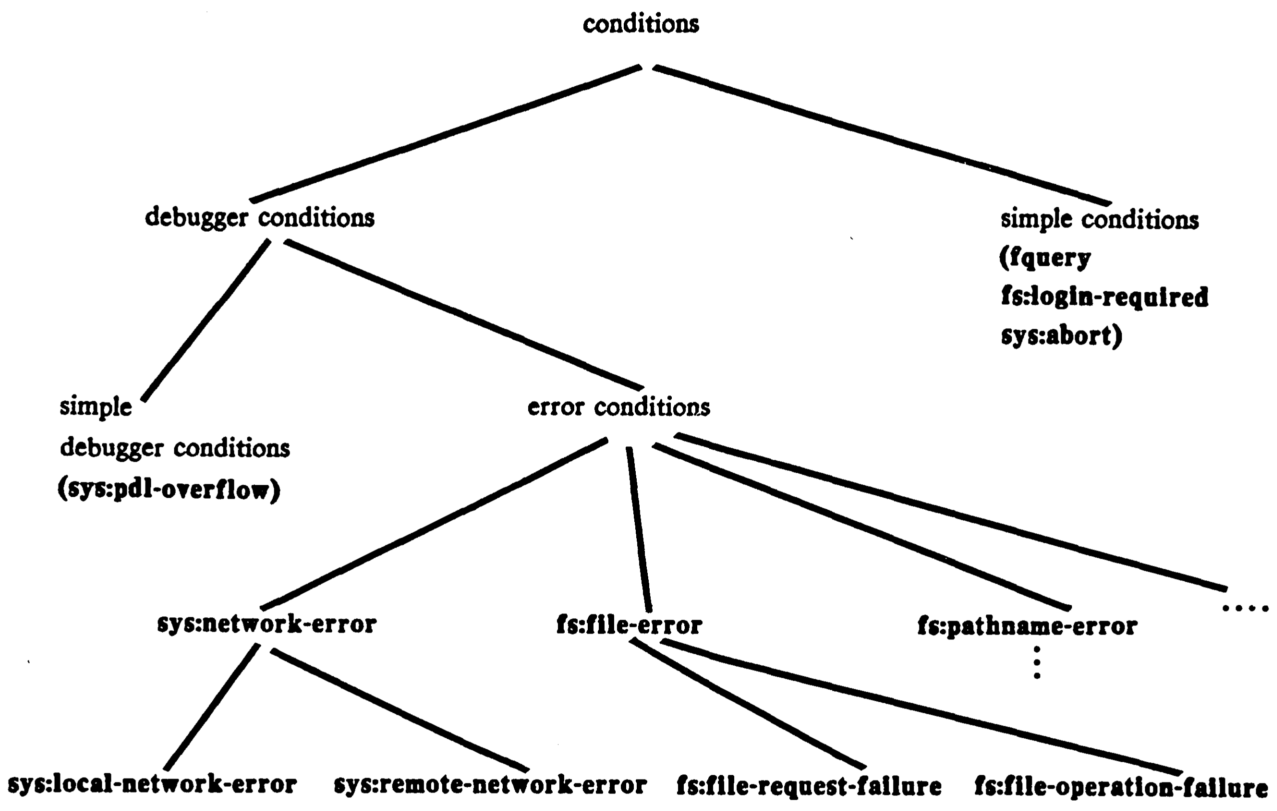


Figure 12. Condition flavor hierarchy

allowing either a handler or the user to proceed from the error. If the variable ***break-on-warnings*** has a **nil** value, **warn** prints its message without entering the Debugger; if ***break-on-warnings*** has a non-**nil** value, **warn** prints its warning message from the Debugger.

22.3 Creating New Conditions

An application might need to detect and signal events that are specific to the application. To support this, you need to define new conditions.

Defining a new condition is straightforward. For simple cases, you need only two forms: one defines the flavor, and the other defines a **dbg:report** method. Build the flavor definition on either **error** or **condition**, depending on whether or not the condition you are defining represents an error. The following example defines an error condition.

```
(defflavor block-wrong-color () (error))

(defmethod (dbg:report block-wrong-color) (stream)
   (format stream "The block was of the wrong color."))
```

Your program can now signal the error as follows:

```
(error 'block-wrong-color)
```

dbg:report requires one argument, which is a stream for it to use in printing an error message. Its message should be a sentence, ending with a period and with no leading or trailing newlines.

The **dbg:report** method must not depend on the dynamic environment in which it is invoked. That is, it should not do any free references to special variables. It should use only its own instance variables. This is because the condition object might receive a **dbg:report** message in a dynamic environment that is different from the one in which it was created. This situation is common with **condition-case**.

The above example is adequate but does not take advantage of the power of the condition system. For example, the error message tells you only the class of event detected, not anything about this specific event. You can use instance variables to make condition objects unique to a particular event. For example, add instance variables **block** and **color** to the flavor so that **error** can use them to build the condition object:

```
(defflavor block-wrong-color (block color) (error)
  :initable-instance-variables
  :readable-instance-variables)
```

```
(defmethod (dbg:report block-wrong-color) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

The **:initable-instance-variables** option defines **:block** and **:color** init options; the **:readable-instance-variables** option defines methods for the **block-wrong-color** flavor, which handlers can call.

Your program would now signal the error as follows:

```
(error 'block-wrong-color :block the-bad-block
                          :color the-bad-color)
```

The only other interesting thing to do when creating a condition is to define proceed types. See the section "Proceeding", page 587.

It is a good idea to use **compile-flavor-methods** for any condition whose instantiation is considered likely, to avoid the need for run-time combination and compilation of the flavor. See the macro **compile-flavor-methods** in *Symbolics Common Lisp: Language Dictionary*. Otherwise, the flavor must be combined and compiled the first time the event occurs, which causes perceptible delay.

22.3.1 Creating a Set of Condition Flavors

You can define your own sets of conditions and condition hierarchies. Just create a new flavor and build the flavors on each other accordingly. The base flavor for the set does not need a **dbg:report** method if it is never going to be signalled itself. For example:

```
(defflavor block-world-error () (error))

(defflavor block-wrong-color (block color) (block-world-error)
  :initable-instance-variables)

(defflavor block-too-big (block container) (block-world-error)
  :initable-instance-variables)

(defmethod (dbg:report block-too-big) (stream)
  (format stream "The block ~S is too big to fit in the ~S."
                 block container))

(defmethod (dbg:report block-wrong-color) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

22.4 Establishing Handlers

22.4.1 What Is a Handler?

A handler consists of user-supplied code that is invoked when an appropriate condition signal occurs. Genera includes default handlers for all standard conditions. Application programs need not handle all conditions explicitly but can provide handlers for just the conditions most relevant to the needs of the application.

22.4.2 Classes of Handlers

There are five classes of handlers, as follows:

- Bound handlers
- Default handlers
- Global handlers
- Interactive handlers
- Restart handlers

22.4.2.1 Bound Handlers

The simplest form of handler handles every error condition, each in the same way. The form for binding this handler is **ignore-errors**. In addition, four basic forms are available to bind handlers for particular conditions. Each of these has a standard version and a conditional variant. In the conditional variants, the handlers are bound only if some expression is true.

22.4.2.2 Basic Forms for Bound Handlers

condition-bind *list* &body *body* — This is the most general form. It allows the handler to run in the dynamic environment in which the error was signalled and to try to proceed from the error.

condition-bind-if *cond-form list* &body *body*
The conditional variant of **condition-bind**.

condition-case *(&rest varlist) form* &rest *clauses*
This is the simplest form to use. It returns to the dynamic environment in which the handler was bound and so does not allow proceeding.

condition-case-if *(cond-form* &rest *varlist) form* &rest *clauses*
The conditional variant of **condition-case**.

condition-call *(*&rest *varlist) form* &body *clauses*
This is a more general version of **condition-case**. It uses user-specified predicates to select the clause to be executed.

condition-call-if *(cond-form* &rest *varlist) form* &body *clauses*
The conditional variant of **condition-call**.

ignore-errors &body *body*
Sets up a very simple handler on the bound handlers list that handles all error conditions. If an error signal occurs while *body* is executing, **ignore-errors** immediately returns with **nil** as its first value and something not **nil** as its second value.

22.4.2.3 Default Handlers

Default handlers are examined by the signalling mechanism only after all of the bound handlers have been examined. Thus, the handlers established by a **condition-bind** which is dynamically outside of a **condition-bind-default** can take precedence over the handlers established by **condition-bind-default**. In all other respects, the binding forms work like those for bound handlers.

22.4.2.4 Basic Forms for Default Handlers

condition-bind-default *list* &body *body*
This is a variant of **condition-bind**. It binds a handler on the default condition list instead of the bound condition list. The distinction is described in these two sections. See the section "Signalling Conditions", page 578. See the section "Default Handlers and Complex Modularity", page 583.

condition-bind-default-if *cond-form list* &body *body*
The conditional variant of **condition-bind-default**.

22.4.2.5 Global Handlers

A global handler is like a bound handler, with an important exception: unlike a bound handler which is of dynamic extent, a global handler is explicitly defined and is of *indefinite* extent. (Whereas **condition-bind** is like *binding* a special variable, **define-global-handler** is like *setting* a special variable.) Once defined, a global handler must, therefore, be specifically removed.

A note of caution: The global handler functions do not maintain the order of the global handler list in any way. If there are two handlers whose conditions overlap each other in such a way that some instantiable condition could be handled by either, then either handler might run, depending on the order in which they were defined.

22.4.2.6 Basic Forms for Global Handlers

define-global-handler *name condition arglist* &body *body*	Defines a global handler function named *name.*
undefine-global-handler *name*	Removes the global handler function named by *name.*
dbg:describe-global-handlers	Displays the list of conditions for which global handlers have been defined, as well as a list of these handlers.

22.4.3 Application: Handlers Examining the Stack

condition-bind handlers are invoked in the dynamic environment in which the error is signalled. Thus the Lisp stack still holds the frames that existed when the error was signalled. A handler can examine the stack using the functions described in this section.

One important application of this facility is for writing error logging code. For example, network servers might need to keep providing service even though no user is available to run the Debugger. By using these functions, the server can record some information about the state of the stack into permanent storage, so that a maintainer can look at it later and determine what went wrong.

These functions return information about stack frames. Each stack frame is identified by a *frame*, represented as a Lisp locative pointer. In order to use any of these functions, you need to have appropriate environment bindings set up. The macro **dbg:with-erring-frame** both sets up the environment properly and provides a frame pointer to the stack frame that got the error. Within the body of that macro, use the appropriate functions to move up and down stack frames; these functions take a frame pointer and return a new frame pointer by following links in the stack.

These frame-manipulating functions are actually *subprimitives*, even though they do not have a % sign in their name. If given an argument that is not a frame pointer, they stand a good chance of crashing the machine. Use them with care.

The functions that return new frame pointers work by going to the *next frame* or the *previous frame* of some category. "Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack). "Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; towards the base of the stack).

These functions assume three categories of frames: *interesting active* frames, *active* frames, and *open* frames.

- An active frame simply means a procedure that is currently running (or active) on the stack.
- Interesting active frames include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as frames for **eval**, **zl:apply**, **funcall**, **let**, and other basic Lisp special forms. The list of such functions is the value of the system constant **dbg:*uninteresting-functions***.
- Open frames include all the active frames as well as frames that are still under construction, for functions whose arguments are still being computed. Open frames and active frames are synonymous on the 3600 series.

22.4.3.1 Functions for Examining Stack Frames

The functions in this section all take a frame pointer, *frame*, as an argument. For functions that indicate a direction on the stack, using **nil** as the argument indicates the frame at the relevant end of the stack. For example, when you are using a function that looks up the stack, **nil** as the argument indicates the top-most stack frame.

Remember to use the functions in this section only within the context of the **dbg:with-erring-frame** macro.

dbg:with-erring-frame *(frame-var condition)* &body *body*
: Sets up an environment with appropriate bindings for using the rest of the functions that examine the stack.

dbg:get-frame-function-and-args *frame*
: Returns a list containing the name of the function for *frame* and the values of the arguments.

dbg:frame-next-active-frame *frame*
: Returns a frame pointer to the next active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-interesting-active-frame *frame*
: Returns a frame pointer to the next interesting active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-open-frame *frame* Returns the frame pointer to the next open frame following *frame*, or **nil** if *frame* is the last open frame on the stack.

dbg:frame-previous-active-frame *frame*
Returns the frame pointer to the previous active frame before *frame*, or **nil** if *frame* is the first active frame on the stack.

dbg:frame-previous-interesting-active-frame *frame*
Returns the frame pointer to the previous interesting active frame before *frame*, or **nil** if *frame* is the first active frame on the stack.

dbg:frame-previous-open-frame *frame*
Returns the frame pointer to the previous open frame before *frame*, or **nil** if *frame* is the first open frame on the stack.

dbg:frame-next-nth-active-frame *frame* &optional *(count* **1***)*
Returns the frame pointer to the next nth active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-nth-interesting-active-frame *frame* &optional *(count* **1***)*
Returns the frame pointer to the next nth interesting active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-nth-open-frame *frame* &optional *(count* **1***)*
Returns the frame pointer to the next nth open frame following *frame*, or **nil** if *frame* is the last open frame on the stack.

dbg:frame-out-to-interesting-active-frame *frame*
Returns the frame pointer to the out to interesting active frame, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-active-p *frame*
Returns **t** if the *frame* is active, or **nil** otherwise.

dbg:frame-real-function *(frame)→(real-function first-interesting-local)*
Returns either the function object associated with *frame*, or **self** when the frame was the result of sending a message to an instance.

dbg:frame-total-number-of-args *frame*
Returns the number of arguments that were passed in *frame*.

dbg:frame-number-of-spread-args *frame* &optional *(type* **:supplied***)*
Returns the number of "spread" arguments that were passed in *frame*. *type* requests more specific definition of the number.

dbg:frame-arg-value *(frame arg-name-or-number* &optional *callee-context no-error-p)→(value location)*
Returns the value of the *n*th argument to *frame*. Returns a second value, which is a locative pointer to the word in the stack that holds the argument. If *n* is out of range, then the function takes action based on *no-error-p*; if the latter is **nil** it signals an error otherwise it returns **nil**.

dbg:frame-number-of-locals *frame* Returns the number of local variables allocated for *frame*.

dbg:frame-local-value *(frame local-name-or-number* &optional *no-error-p)→(value location)*
Returns the value of the *n*th local variable in *frame*. Returns a second value, which is a locative pointer to the word in the stack that holds the local variable.

dbg:frame-self-value *frame* &optional *instance-frame-only*
Returns the value of **self** in *frame*, or **nil** if **self** does not have a value.

dbg:frame-real-value-disposition *frame*
Returns a symbol indicating how the calling function is going to handle the values to be returned by this frame. If the calling function just returns the values to its caller, then the symbol indicates how the final recipient of the values is going to handle them.

dbg:print-function-and-args *frame* &optional *show-pc-p show-source-file-p show-local-if-different present-as-function*
Prints the name of the function executing in *frame* and the names and values of its arguments, in the same format as the Debugger uses. If *show-pc-p* is true, the program counter value of the frame, relative to the beginning of the function, is printed in octal.

dbg:print-frame-locals *frame local-start* &optional *(indent 0)*

Prints the names and values of the local variables of *frame*.

22.5 Signalling Conditions

22.5.1 Signalling Mechanism

The following functions and macros invoke the signalling mechanism, which finds and invokes a handler for the condition.

error
signal
cerror
zl:ferror
zl:fsignal
signal-proceed-case

22.5.1.1 Finding a Handler

The signalling mechanism finds a handler by inspecting five lists of handlers in the following order:

1. It first looks down the list of *bound* handlers, which are handlers set up by **condition-bind, condition-case,** and **condition-call** forms.

2. Next, it looks down the list of *default* handlers, which are set up by **condition-bind-default.**

3. Next, it looks down the list of *global* handlers, which are set up by **define-global-handler.**

4. Next, it looks down the list of *interactive* handlers. This list normally contains only one handler, which enters the Debugger if the condition is based on **dbg:debugger-condition** and declines to handle it otherwise.

5. Finally, it looks down the list of *restart* handlers, which are set up by **error-restart, error-restart-loop,** and **catch-error-restart.** See the section "Default Handlers and Complex Modularity", page 583. See the section "Restart Handlers", page 585.

If it gets to the end of the last list without finding a willing handler, one of two things happens.

- **signal** returns **nil** when both of the following are true:

  - ° The condition was signalled with **signal**, **cerror**, **signal-proceed-case**, or **zl:fsignal**.
  - ° The condition object is not an instance of a condition based on **error**.

- The Debugger assumes control.

The signalling mechanism checks each handler to see if it is willing to handle the condition. Some handlers have the ability to decline to handle the condition, in which case the signalling mechanism keeps searching. It calls the first willing handler it finds.

As we have seen, the signalling mechanism searches for handlers in a specific order. It looks at all the bound handlers before any of the default handlers and all of the default handlers before any of the restart handlers. Thus, it tries any **condition-bind** handler before any handler bound by **condition-bind-default**, even though the **condition-bind-default** is within the dynamic scope of the **condition-bind**. Similarly, it considers a **condition-bind** handler before an **error-restart** handler, even when the **error-restart** handler was bound more recently. See the section "Default Handlers and Complex Modularity", page 583.

While a bound or default handler is executing, that handler and all handlers inside it are removed from the list of bound or default handlers. This is to prevent infinite recursion when a handler signals the same condition that it is handling, as in the following simplistic example:

```
(condition-bind ((error #'(lambda (x) (error "foo"))))
  (error "foo"))
```

If you want recursion, the handler should bind its own condition.

22.5.1.2 Signalling Simple Conditions

If a simple condition or a debugger condition not based on **error** is signalled, the signalling mechanism searches for a handler on the bound handler and default handler lists. When it finds one, it invokes it. Otherwise, the signalling mechanism checks for a global handler and invokes it if found. If there are no global handlers, the signalling mechanism checks for an interactive handler, invoking the first one it finds. If there are no interactive handlers, the first restart handler for that condition is invoked. If no restart handler for the condition is found, **signal** returns **nil**; **error** enters the Debugger.

Normally, there is only one interactive handler. This handler calls the Debugger if the condition is a debugger condition and not a simple condition. See the section "Interactive Handlers", page 584.

22.5.1.3 Signalling Errors

In practice, if the **signal** function is applied to an error condition object, **signal** is very unlikely to return **nil**, because most processes contain a restart handler that handles all error conditions. The function at the base of the stack of most processes contains a **catch-error-restart** form that handles **error** and **sys:abort**. Thus, if you are in the Debugger as a result of an error, you can always use ABORT. The restart handler at the base of the stack always handles **sys:abort** and either terminates or restarts the process.

22.5.1.4 Restriction Due to Scope

A condition must be signalled only in the environment in which the event that it represents took place, to insure that handlers run in the proper dynamic environment. Therefore, you cannot signal a condition object that has already been signalled once. In particular, when you are writing a handler, you cannot have that handler signal its condition argument. Similarly, if a condition object is returned by some program (such as the **open** function given **nil** for the **:error** keyword), you cannot signal that object.

It is not correct to pass on the condition by signalling the handler's condition argument. This is incorrect:

```
(defun condition-handler (condition)
    (if something (throw ...) (signal condition)))
```

Instead you should do this:

```
(defun condition-handler (condition)
    (if something (throw ...) nil))
```

or this:

```
(defun condition-handler (condition)
    (if something (throw ...) (signal 'some-other-condition)))
```

22.5.2 Condition-Checking and Signalling Functions and Variables

signal *flavor* &rest *init-options*

This is the primitive function for signalling a condition. **signal** creates a new condition object of the specified *flavor* and signals it.

error {*format-string* rest *format-args*} or {*condition* &rest *init-options*} or {*condition-object*}

Signals a condition that is not proceedable. In the simplest case, signals a **zl:ferror** condition. If called with *condition* &rest **cl:init-options**, creates a condition of type *condition* and signals it. If no handler for the condition exists, the debugger assumes control whether or not the object is an error object. In its most advanced form **error** is called with a single argument, *condition-object* and *init-options* is ignored. **error** never returns to its caller.

cerror *optional-condition-name continue-format-string error-format-string* &rest *args*
Signals a proceedable error and enters the debugger.

warn *optional-options optional-condition-name format-string* &rest *format-args*
Prints a warning message and does not enter the debugger if ***break-on-warnings*** is **nil**, otherwise enters the debugger, prints the message, and allows user to proceed.

break-on-warnings
If value of this variable is **nil**, **warn** prints its warning message without entering the Debugger; if the value is not **nil**, **warn** enters the Debugger and prints the warning message. Default is **nil**.

catch-error *form* &optional (*printflag* **t**)
Evaluates *form*, trapping all errors. If the value of *printflag* is not **nil** and an error occurs during evaluation, the function prints an error message and returns **t**. If no error occurred, the value of *form* and **nil** are returned. *form* is not evaluated for multiple values.

make-condition *condition-name* &rest *init-options*
Creates a condition object of the specified *condition-name* with the specified *init-options*. This object can then be signalled by passing it to **signal** or **error**.

zl:fsignal *format-string* &rest *format-args*
This is a simple function for signalling when you do not care to use a particular condition. **zl:fsignal**, signals **dbg:proceedable-ferror**. Obsolete; use **cerror** instead.

zl:ferror *format-string* &rest *format-args*
This is a simple function for signalling when you do not care what the condition is. **zl:ferror** signals **zl:ferror**. Obsolete; use **error** instead.

zl:parse-ferror *format-string* &rest *format-args*
Signals an error of flavor **zl:parse-ferror**.

errorp *thing*
Returns **t** if *thing* is an error object, and **nil** otherwise.

check-type *place type* &optional *(type-string* **'nil***)*
Signals a proceedable error if the contents of *place* are not of the desired type. Accepts replacement value for *place*.

assert *test-form* &optional *references format-string* &rest *format-args*
Signals a proceedable error if the value of *test-form* is **nil**. Accepts replacement values for variables in *test-form*.

ccase *object* &body *body*
"Continuable exhaustive case." Like **case**, selects one of its clauses for execution by comparing a value to various constants, but does not allow an explicit **t** clause. Signals a proceedable error if no clause is satisfied; accepts replacement value for *object*.

ecase *object* &body *body*
"Exhaustive case," or "error-checking case." Like **case**, selects one of its clauses for execution by comparing a value to various constants, but does not allow an explicit **t** clause. Signals an error if no clause is satisfied. It is not permissible to continue from this error.

ctypecase *object* &body *body*
"Continuable exhaustive type case." Like **typecase**, selects one of its clauses by examining the type of an object, but does not allow an explicit **otherwise** or **t** clause. Signals a proceedable error if no clause is satisfied. Accepts replacement value for *object*.

etypecase *object* &body *body*
"Exhaustive type case," or "error-checking type case." Like **typecase**, but does not allow an explicit **otherwise** or **t** clause. Signals an error if no clause is satisfied. It is not permissible to continue from this error.

zl:check-arg *arg-name predicate-or-form type-string*
Checks arguments to make sure they are valid. Accepts replacement value for *arg-name*.

zl:check-arg-type *arg-name type* &optional *type-string*
A useful variant of **zl:check-arg**.

zl:argument-typecase *arg-name* &body *clauses*
This is a hybrid of **zl:typecase** and **zl:check-arg-type**. Automatically generates an **otherwise** clause that signals an error. Accepts replacement value for *arg-name*.

22.6 Default Handlers and Complex Modularity

When more than one handler exists for a condition, which one should be invoked? The signalling mechanism has an elaborate rule, but in practice, it usually invokes the innermost handler. See the section "Finding a Handler", page 578. "Innermost" is defined dynamically and thus means "the most recently bound handler".

This decision is made on the basic principle of modularity and referential transparency: a function should behave the same way, regardless of what calls it. Therefore, whether a handler bound by a function gets invoked should not depend on what is going on with that function's callers.

For example, suppose function **a** sets up a handler to deal with the **fs:file-not-found** condition, and then calls procedure **b** to perform some service for it. Now, unbeknownst to **a**, **b** sometimes opens a file, and **b** has a condition handler for **fs:file-not-found**. If **b**'s file is not found, **b**'s handler handles the error rather than **a**'s. This is as it should be, because it should not be visible to **a** that **b** uses a file (this is a hidden implementation detail of **b**). **a**'s unrelated condition handler should not meddle with **b**'s internal functioning. Therefore, the signalling mechanism follows a basic inside-to-outside searching rule.

Sometimes a function needs to signal a condition but still handle the condition itself if none of its callers handles it. On first encounter, this need seems to require an outside-to-inside searching rule instead of the inside-to-outside searching rule mandated by modularity considerations. How can you circumvent the rules to allow a function to handle something only if no outer function handles it?

Several strategies are available for dealing with this. Genera provides several mechanisms in order to allow experimentation and flexibility.

- The simplest solution is to provide a proceed type for proceeding from the Debugger. That is, your program signals an error to allow callers to handle the condition. If none of them handles it, the Debugger assumes control. Provided that the user decides to use the proceed type, your program then gets to handle the condition. If what your program wanted to do was to prompt the user anyway, this might be the right thing. This is most likely true if you think that a program error is probably happening and the user might want to be able to trace and manipulate the stack using the Debugger.

- Another simple solution is to signal a condition that is not an error. **signal** returns **nil** when no handler is found, and your program can take appropriate action.

- Use **condition-bind-default** to create a handler on the default handler list. The signalling mechanism searches this list only after searching through all regular bound handlers. One drawback of this scheme is that it works only to one level. If you have three nested functions, you cannot get outside-to-inside modularity for all three, because only two lists exist, the bound list and the default list. This facility is probably good enough for some applications however.

- Use **dbg:condition-handled-p** to determine whether a handler has been bound for a particular flavor. This has the advantage that it works for any number of levels of nested handler, instead of only two. One disadvantage is that it can return **:maybe**, which is ambiguous.

The simple solutions work only if your program is doing the signalling. If some other program is signalling a condition, you cannot control whether the condition is an error condition or whether it has any proceed types; you can only write handlers.

22.7 Interactive Handlers

The interactive handler list contains one element: a handler that invokes the Debugger if the condition is built on **dbg:debugger-condition** and declines to handle the condition if it is not. No standard procedure exists for changing the contents of this list.

One of the original design goals of the condition signalling mechanism was to support building complex applications that could take over the function of the Debugger and provide their own. The exact definition of the problem is not completely clear however. We are not sure whether the current system provides this functionality.

If you are writing an application that needs to take over error handling completely, you might be able to create a **condition-bind** handler that handles **error**, to prevent invocation of the Debugger. This strategy might have problems that we have not anticipated. If you really need to get the Debugger out of the way, you might try changing the interactive handler list. We have not defined a way to do this; read the code for complete details. We cannot guarantee that whatever you do will work in future releases. However, we encourage your experimentation. Please contact us so that we can help you if possible.

Briefly, the variable holding the list is named **dbg:*interactive-handlers***, which holds an interactive handler object. The list is reset to hold the standard Debugger when you warm boot the machine.

Interactive Handler Messages

:handle-condition-p *cond*	Examines the condition object, *cond*. Returns **nil** if it declines to handle the condition and something other than **nil** when it is prepared to handle the condition.
:handle-condition *cond ignore*	Examines the condition object, *cond*; you should handle this condition, ignoring the second argument. This form can return values or **throw** in the same way that **condition-bind** handlers can.

22.8 Restart Handlers

One way to handle an error is to restart at some earlier point the program that got the error. A program can specify points where it is safe or convenient for it to be restarted should a condition signal occur during processing a function. The basic special form for doing this is called **error-restart**. The following example is taken from the system code:

```
(defun connect (address contact-name
                &optional (window-size default-window-size)
                (timeout (* 10. 60.))
                &aux conn real-address (try 0))
  (error-restart (connection-error
                   "Retry connection to ~A at ~S with longer timeout"
                   address contact-name)
    forms...))
```

This code fragment evaluates *forms* and returns the final value(s) if successful. If the Debugger assumes control as a result of a **sys:connection-error** condition, the user is given the opportunity of restarting the program. The Debugger's prompt message would be something like this:

```
s-A:  Retry connection to SCRC at FILE 1 with longer timeout
```

If the user were to press s-A at this point, the forms implementing the connection would be evaluated again. That is, the body of the **error-restart** would be started again from the beginning.

Two variations on this basic paradigm are provided. **error-restart-loop** is an infinite loop version of **error-restart**. It always starts over regardless of whether a condition has been signalled. **catch-error-restart** never restarts, even when a condition is signalled. Instead it always returns, returning either the values from the body (if successful) or **nil** if a condition signal occurred.

catch-error-restart is the most primitive version of this control structure. The other two are built from it. It too has a conditional variant, **catch-error-restart-if**, for binding a restart handler conditionally.

A common paradigm is to use one of these forms in the command loop of an interactive program, with *condition-flavor* being the list **(error sys:abort)**. This way, if an unhandled error occurs, the user is offered the option of returning to the command loop, and the ABORT key returns to the command loop. Which form you use depends on the nature of your command loop.

22.8.1 Restart Functions

The use of "error-" in the names of these functions has no real significance. They could have been called **cl:condition-restart**, **cl:condition-restart-loop**, and so on, because they apply to all conditions.

error-restart *(condition-flavor format-string . format-args)* &body *body*

The basic special form for a program to specify safe restart points. Establishes a restart handler for *flavors* then evaluates *body*. If restart handler is not invoked, **error-restart** returns the values produced by the last form in *body*. If restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart** form and execution of *body* starts all over again.

error-restart-loop *(condition-flavor format-string . format-args)* &body *body*

Establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **error-restart-loop** evaluates the body again and again, in an infinite loop. Use the **return** function to leave the loop. This mechanism is useful for interactive top levels. If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart-loop** form and execution of the body is started all over again.

catch-error-restart *(condition-flavor format-string . format-args)*

Establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **catch-error-restart** returns the values produced by the last form in the body, and the restart handler disappears. If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment of the **catch-error-restart** form. In this case, **catch-error-restart** also returns **nil** as its first value and something other than **nil** as its second value.

catch-error-restart-if *cond-form (condition-flavor format-string . format-args)* &body *body*
The conditional variant of **catch-error-restart**. This form establishes its restart handler conditionally.

22.8.2 Invoking Restart Handlers Manually

Function **dbg:invoke-restart-handlers** searches the list of restart handlers to find a restart handler. The first handler that it finds to handle the condition is invoked. The function returns **nil** if no appropriate restart handler is found.

dbg:invoke-restart-handlers can be called by handlers set up by **condition-bind** or **condition-bind-default**. The *object* argument should be the condition object passed to the handler. The handler calls this function to bypass the interactive handlers list, letting the innermost restart handler handle the condition. A program that wants to attempt to continue with a computation in the presence of errors might find this useful. For example, it could be used to support batch-mode compilation, with the user away from the console.

22.9 Proceeding

In some situations, execution can proceed past the point at which a condition was signalled. Events for which this is the case are called *proceedable conditions.* Some external agent makes the decision about whether it is reasonable to proceed after repairing the original problem. The agent is either a **condition-bind** handler or the user operating the Debugger.

In general, many different ways are available to proceed from a particular condition. Each way is identified by a *proceed type,* which is represented as a symbol. Condition objects created with **error** instead of **signal** do not have any proceed types.

22.9.1 Protocol for Proceeding

For proceeding to work, two conceptual agents must agree:

- The programmer who wrote the program that signals the condition;
- The programmer who wrote the **condition-bind** handler that decided to proceed from the condition, or else the user who told the Debugger to proceed.

The signaller signals the condition and provides the various proceed types. The handler chooses from among the proceed types to make execution proceed.

Each agent has certain responsibilities to the other; each must follow the protocol described below to make sure that any handler interacts correctly with any signaller. The following description should be considered a two-part protocol that each agent must follow in order to communicate correctly with the other.

In very simple cases, the signaller can use **cerror**, which does not require any new flavor definitions.

In all other cases, the signaller signals the condition using **signal** or **signal-proceed-case**. The signaller also defines a condition flavor with at least one method to handle a proceed type. New proceed types are always defined by adding a new case to the **sys:proceed** method (which is defined to use **:case** method combination) to the condition flavor. The method must always return values rather than throwing.

In **:case** method combination, the first argument to the **sys:proceed** method is like a subsidiary message name, causing a further dispatch just as the original message name caused a primary dispatch.

Here's an example from the system:

```
(defmethod (sys:proceed sys:subscript-out-of-bounds  :new-subscript)
           (&optional (sub (prompt-and-read :number
                                            "Subscript to use instead: ")))
  "Supply a different subscript."
  (values :new-subscript sub))
```

This code fragment creates a proceed type called **:new-subscript** for the condition flavor **sys:subscript-out-of-bounds**.

To proceed from a condition, a handler function calls the **sys:proceed** generic function with one or more arguments. The first argument is the *condition* object. The second argument is the proceed type, and any remaining arguments are the arguments for that proceed type.

The condition flavor defined by the program signalling the error defines the proceed types that are available to **sys:proceed** for a particular condition. You can also define a method that creates a new proceed type with the **:case** method combination.

All of the arguments to a **sys:proceed** method must be optional arguments. The **sys:proceed** method should provide default values for all its arguments. One useful way of doing this is to prompt a user for the arguments using the ***query-io*** stream. The example uses **prompt-and-read**. If all the optional arguments were supplied, the **sys:proceed** method must not do any input or output using ***query-io***.

This facility has been defined assuming that **condition-bind** handlers would supply all the arguments for the method themselves. The Debugger runs this method and does not supply arguments, relying on the method to prompt the user for the arguments.

As in the example, the method should have a documentation string as the first form in its body. The **dbg:document-proceed-type** generic function for a proceedable condition object displays the string. This string is used by the Debugger as a prompt to describe the proceed type. For example, the subscript example might result in the following Debugger prompt:

```
s-A: Supply a different subscript
```

The string should be phrased as a one-line description of the effects of proceeding from the condition. It should not have any leading or trailing newlines. (You can use the messages that the Debugger prints out to describe the effects of the `s-` commands as models if you are interested in stylistic consistency.)

Sometimes a simple fixed string is not adequate. You can provide a piece of Lisp code to compute the documentation text at run time by providing your own method for **dbg:document-proceed-type**. This method definition takes the following form:

```
(defmethod (dbg:document-proceed-type condition-flavor proceed-type)
           (stream)
  body...)
```

The body of the method should print documentation for *proceed-type* of *condition-flavor* onto *stream*.

The body of the **sys:proceed** method can do anything it wants. In general, it tries to repair the state of things so that execution can proceed past the point at which the condition was signalled. It can have side-effects on the state of the environment, it can return values so that the function that called **signal** can try to fix things up, or it can do both. Its operation is invisible to the handler; the signaller is free to divide the work between the function that calls **signal** and the **sys:proceed** method as it sees fit. When the **sys:proceed** method returns, **signal** returns all of those values to its caller. That caller can examine them and take action accordingly.

The meaning of these returned values is strictly a matter of convention between the **sys:proceed** method and the function calling **signal**. It is completely internal to the signaller and invisible to the handler. By convention, the first value is usually the name of a proceed type. See the section "Signallers", page 591.

A **sys:proceed** method can return a first value of **nil** if it declines to proceed from the condition. If a **nil** returned by a **sys:proceed** method becomes the return value for a **condition-bind** handler, this signifies that the handler has declined to handle the condition, and the condition continues to be signalled. When the **sys:proceed** function is called by the Debugger, the Debugger prints a message saying that the condition was not proceeded, and it returns to its command level. This might be used by an interactive **sys:proceed** method that gives the user the opportunity either to proceed or to abort; if the user aborts, the method returns **nil**. Returning **nil** from a **sys:proceed** method should not be used as a substitute for detecting earlier (such as when the condition object is created) that the proceed type is inappropriate for that condition.

22.9.2 Proceed Type Functions

By default, condition objects have to handle all proceed types defined for the condition flavor. Condition objects can be created that handle only some of the proceed types for their condition flavor. When the signaller creates the condition object (with **signal** or **make-condition**), it can use the **:proceed-types** init option to specify which proceed types the object accepts. The value of the init option is a list of keyword symbols naming the proceed types.

```
(signal 'my-condition :proceed-types '(:abc))
```

The **dbg:proceed-types** generic function for a condition object returns a list of keywords for the proceed types that the object is prepared to handle.

The **dbg:proceed-type-p** generic function examines the list of valid proceed types to see whether it contains a particular proceed type.

A condition flavor might also have an **:init** daemon that could modify its **dbg:proceed-types** instance variable.

22.9.3 Proceeding With condition-bind Handlers

Suppose the handler is a **condition-bind** handler function. Just to review, when the condition is signalled, the handler function is called with one argument, the condition object. The handler function can throw to some tag, return **nil** to say that it doesn't want to handle the condition, or try to proceed the condition.

The handler must not attempt to proceed using an invalid proceed type. It must determine which proceed types are valid for any particular condition object. It must do this at run-time because condition objects can be created that do not handle all of the proceed types for their condition flavor. See the init option **(flavor:method :proceed-types condition)** in *Symbolics Common Lisp: Language Dictionary*.

In addition, condition objects created with **error** instead of **signal** do not have any proceed types. The handler can use the **dbg:proceed-types** and **dbg:proceed-type-p** functions to determine which proceed types are available.

To proceed from a condition, a handler function calls **sys:proceed** on the condition object with one or more additional arguments. The first additional argument is the proceed type (a keyword symbol) and the rest are the arguments for that proceed type. All of the standard proceed types are documented with their condition flavors. Thus the programmer writing the handler function can determine the meanings of the arguments. The handler function must always pass all of the arguments, even though they are optional.

Calling **sys:proceed** should be the last thing the handler does. It should then return immediately, propagating the values from the **sys:proceed** method back to its caller. Determining the meaning of the returned values is the business of the signaller only; the handler should not look at or do anything with these values.

22.9.4 Proceed Type Names

Any symbol can be used as the name of a proceed type, although using keyword symbols is conventional. The symbols **:which-operations** and **:case-documentation** are not valid names for proceed types because they are treated specially by the **:case** flavor combination. Do not use either of these symbols as the name of a proceed type when you create a new condition flavor.

22.9.5 Signallers

Signallers can use the **signal-proceed-case** special form to signal a proceedable condition. **signal-proceed-case** assumes that the first value returned by every proceed type is the keyword symbol for that proceed type. This convention is not currently enforced.

22.9.5.1 Example of User-Defined Error Flavor

Here is a nice real-world example of an application program defining its own error flavor, which indicates that an editor buffer was not found, with two proceed types: one proceed type asks the user to supply a new buffer name; the other offers to create a new buffer.

```
(defflavor buffer-not-found ((pathname nil) (name nil)
                             defaults create-p load-p) (error)
  :initable-instance-variables
  :readable-instance-variables)
```

```
(defmethod (dbg:report buffer-not-found) (stream)
  (format stream "The buffer ~:[named~;for pathname~]
                              /"~A/" was not found."
                              (null name) (or name pathname)))

(defmethod (sys:proceed buffer-not-found :choose-another-buffer)
           (&optional new-name)
  (if (not new-name)
      (setq new-name
            (cond (name (prompt-and-read ':string
               "Supply another buffer name to use instead of ~A:  "
                                        name))
                  (t (prompt-and-read ':pathname
                                      "Supply another pathname to
                    use instead of ~A~ ~:[~; (Default = ~A)~]:  "
                                      name defaults
                                      (fs:default-pathname
                                        defaults)))))
      (find-or-create-buffer nil (and name new-name)
                             (and pathname (fs:parse-pathname
                                              new-name))
                             nil nil
                             defaults
                             create-p load-p nil)))

(defmethod (dbg:document-proceed-type buffer-not-found
                                      :choose-another-buffer)
           (stream)
  (format stream
          "Supply another ~:[buffer name~;pathname~]
            to use instead."
          (null name)))

(defmethod (sys:proceed buffer-not-found :create-buffer) ()
  (find-or-create-buffer nil name pathname nil nil
                         defaults t load-p nil))
```

```
(defmethod (dbg:document-proceed-type buffer-not-found
                                      :create-buffer)
           (stream)
  (if (null name)
      (format stream "Create a buffer for file /"~A/"
                     ~:[~; and load it from the file~]."
              pathname load-p)
      (format stream "Create a buffer named /"~A/"." name)))

(compile-flavor-methods buffer-not-found)
```

22.10 Issues for Interactive Use

22.10.1 Tracing Conditions

The variable **sys:trace-conditions** is provided for debugging purposes only. It lets you trace the signalling of any condition so that you can figure out what conditions are being signalled and by what function. You can set this variable to **error** to trace all error conditions, for example, or you can be more specific.

You can also customize the error message displayed by the debugger by binding the variable **sys:error-message-hook** to a function which prints what you want. When the debugger finds the value of **sys:error-message-hook**, to be non-**nil**, it **funcalls** (applies the function) with no arguments, and displays the results at the end of its error message display.

22.10.2 Breakpoints

The functions **breakon** and **unbreakon** can be used to set breakpoints in a program. They use the encapsulation mechanism like **trace** and **advise** to force the function to signal a condition when it is called.

breakon sets a breakpoint for the *function-spec*.

condition-form can be used for making a conditional breakpoint. It is evaluated when the function is called. If it returns **nil**, the function call proceeds without signalling anything. *condition-form* is evaluated in the dynamic environment of the function call. You can inspect the arguments of *function-spec* by looking at the variable **arglist**.

unbreakon turns off a breakpoint set by **breakon**. If *function-spec* is not provided, all breakpoints set by **breakon** are turned off. If *condition-form* is provided, it turns off only that condition, leaving any others. If *condition-form* is not provided, the entire breakpoint is turned off for that function. See the section "Encapsulations", page 263.

Calling a function for which a breakpoint is set signals a condition with the following message:

```
Break on entry to function name
```

It provides a **:no-action** proceed type, which allows the function entry to proceed. The "trap on exit" bit is set in the stack frame of the function call, so that when the function returns or is thrown through another condition is signalled. Similarly, the "Break on exit from marked frame" message and the **:no-action** proceed type are provided, allowing the function return to proceed.

22.10.2.1 Breakpoint Functions

breakon &optional *function (condition* **t***)*

Sets a breakpoint for *function*. When *function* is called, the condition **dbg:breakon-trap** is signalled and the Debugger assumes control. With no arguments, **breakon** returns a list of all functions with breakpoints set by **breakon**.

unbreakon &optional *function (condition* **t***)*

Turns off a breakpoint set for *condition*. With no arguments, turns off all breakpoints set by **breakon**.

22.10.3 Debugger Bug Reports

The :Mail Bug Report (c-M) command in the Debugger sends a bug report, creating a new process which by default sends the bug report to the **bug-lispm** mailing list. Also by default, the mail-sending text buffer initially contains a standard set of information dumped by the Debugger. You can control this behavior for your own condition flavors. You can control the mailing list to which the bug report is sent by defining your own primary method for the following message.

You can control the initial contents of the mail-sending buffer by altering the handling of the following message, either by providing your own primary method to replace the default message, or by defining a **:before** or **:after** daemon to add your own specialized information before or after the default text.

22.10.3.1 Debugger Bug Report Functions

dbg:bug-report-recipient-system *condition*

Returns the mailing list to which to send the bug report mail.

dbg:bug-report-description *condition stream nframes*

Prints out the text that is the initial contents of the mail-sending buffer.

22.10.4 Debugger Special Commands

When the Debugger assumes control because an error condition was signalled and not handled, it offers the user various ways to proceed or to restart. Sometimes you want to offer the user other kinds of options. In the system, the most common example of this occurs when you forget to type a package prefix. It signals a **sys:unbound-symbol** error and offers to let you use the symbol from the right package instead. This is neither a proceed type nor a restart-handler; it is a Debugger special command.

You can add one or more special commands to any condition flavor. For any particular instance, you can control whether to offer the special command. For example, the package-guessing service is not offered unless some other symbol with the same print name exists in a different package. Special commands are called only by the Debugger; **condition-bind** handler functions never see them.

Special commands provide the same kind of functionality that a **condition-bind** handler does. There is no reason, for example, that the package-prefix service could not have been provided by **condition-bind.** It is only a matter of convenience to have it in a special command.

To add special commands to your condition flavor, you must mix in the flavor **dbg:special-commands-mixin,** which provides both the instance variable **dbg:special-commands** and several method combinations. Each special command to a particular flavor is identified by a keyword symbol, just the same way that proceed types are identified. You can then create handlers for any of the following messages:

22.10.4.1 Debugger Special Command Functions

dbg:special-command *condition* &rest *pre-command-args*

Sent when the user invokes the special command.

dbg:document-special-command *condition special-command*

Prints the documentation of *special-command.*

dbg:initialize-special-commands *condition*

The Debugger calls this after it prints the error message. The methods for this generic function can remove items from the list

dbg:special-commands in order to decide not to offer these special commands.

22.10.5 Special Keys

The system normally handles the ABORT and SUSPEND keys so that ABORT aborts what you are doing and SUSPEND enters a breakpoint. Without a CONTROL modifier, such a keystroke command takes effect only when the process reads the character from the keyboard; with the CONTROL modifier, a keystroke command takes effect immediately. The META modifier means "do it more strongly"; m-ABORT resets the process entirely, and m-SUSPEND enters the Debugger instead of entering a simple read-eval-print loop.

A complete and accurate description of what these keys do requires a discussion of conditions and the Debugger.

With no CONTROL modifier, ABORT and SUSPEND are detected when your process tries to do input from the keyboard (typically by doing an input stream operation such as **:tyi** on a window). Therefore, if your process is computing or waiting for something else, the effects of the keystrokes are deferred until your process tries to do input.

With a CONTROL modifier, ABORT and SUSPEND are intercepted immediately by the Keyboard Process, which sends your process an **:interrupt** message. Thus, it performs the specified function immediately, even if it is computing or waiting.

ABORT

Prints the following string on the ***terminal-io*** stream, unless it suspects that output on that stream might not work.

```
[Abort]
```

It then signals a **(process-abort *current-process*)**, which is a simple condition. Programs can set up bound handlers for **sys:abort**, although most do not. Many programs set up restart handlers for **sys:abort**; most interactive programs have such a handler in their command loops. Therefore, ABORT usually restarts your program at the innermost command loop. Inside the Debugger, ABORT has a special meaning.

m-ABORT

Prints the following string on the ***terminal-io*** stream, unless it suspects that output on that stream might not work.

```
[Abort all]
```

It then sends your process a **:reset** message, with the argument **:always**. This has nothing to do with condition signalling. It just resets the process completely, unwinding its entire stack. What the process does after that depends on what kind of process it is and how it was created: it might start over from its initial function, or it might disappear. See the section "Processes" in *Internals, Processes, and Storage Management.*

SUSPEND	Calls the **zl:break** function with the argument **zl:break**. This has nothing to do with condition signalling. Pressing the RESUME key causes the process to resume execution. See the special form **zl:break** in *User's Guide to Symbolics Computers.*
m-SUSPEND	Causes the Debugger to assume control without signalling any condition. The RESUME key in the Debugger causes the Debugger to return and the process to resume what it was doing.

Several techniques are available for overriding the standard operation of ABORT and SUSPEND when they are being used with modifier keys.

- For using these keys with the CONTROL modifier, use the asynchronous character facility. See the section "Asynchronous Characters" in *Programming the User Interface, Volume B.*

- Defining your own hook function and binding **tv:kbd-tyi-hook** to it also overrides the interception of these characters with no CONTROL modifier. See the section "Windows as Input Streams" in *Programming the User Interface, Volume B.*

At the Debugger command loop, ABORT is the same as the Debugger :Abort (c-Z) command. It throws directly to the innermost restart handler that is appropriate for either the current error or the **sys:abort** condition.

When the Debugger assumes control, it displays a list of commands appropriate to the current condition, along with key assignments for each. Proceed types come first, followed by special commands, followed by restart handlers. One alphabetic key with the SUPER modifier is assigned to each command on the list. In addition, ABORT is always assigned to the innermost restart handler that handles **sys:abort** or the condition that was signalled; RESUME is always assigned to the first proceed type in the **dbg:proceed-types** list. See the section "Proceed Type Functions", page 590.

If RESUME is not otherwise used, it invokes the first error restart that does not handle **sys:abort**. When you enter the Debugger with m-SUSPEND, RESUME resumes the process.

You can customize the Debugger, assigning certain keystrokes to certain proceed types or special commands, by setting the variables listed below in your init file:

22.10.5.1 Debugger Special Key Variables

dbg:*proceed-type-special-keys* The value of this variable should be an association list associating proceed types with characters. When an error supplies any of these proceed types, the Debugger assigns that proceed type to the specified key.

dbg:*special-command-special-keys*
The value of this variable should be an association list associating proceed types with characters. When an error supplies any of these special commands, the Debugger assigns that special command to the specified key.

22.11 Condition Flavors Reference

A condition object is an instance of any flavor built out of the **condition** flavor. An error object is an instance of any flavor built out of the **error** flavor. The **error** flavor is built out of the **dbg:debugger-condition** flavor, which is built out of the **condition** flavor. Thus, all error objects are also condition objects.

Every flavor of condition that is instantiated must handle the **dbg:report** generic function. (Flavors that just define sets of conditions need not handle it). This message takes a stream as its argument and prints out a textual message describing the condition on that stream. The printed representation of a condition object is like the default printed representation of any instance when slashifying is turned on. However, when slashifying is turned off (by **princ** or the **~A format** directive), the printed representation of a condition object is its **dbg:report** method. Example:

```
(condition-case (co)
    (open "f:>a>b.c")
  (fs:file-not-found
    (prin1 co)))     prints out  #<QFILE-NOT-FOUND 33712233>

(condition-case (co)
    (open "f:>a>b.c")
  (fs:file-not-found
    (princ co)))     prints out  The file was not found
                                 For F:>a>b.c
```

22.11.1 Generic Functions and Init Options

These functions can be sent to any condition object. They are handled by the basic **condition** flavor, on which all condition objects are built. Some particular condition flavors handle other methods; those are documented along with the particular condition flavors in another section. See the section "Standard Conditions", page 600.

22.11.1.1 Basic Condition Methods and Init Options

dbg:document-proceed-type *condition proceed-type stream*

Prints out a description of what it means to proceed, using the given *proceed-type*, from this condition, on *stream*.

dbg:proceed-type-p *condition proceed-type*

Returns **t** if *proceed-type* is one of the valid proceed types of this condition object. Otherwise, returns **nil**.

dbg:proceed-types *condition*

Returns a list of all the valid proceed types for this condition.

dbg:set-proceed-types *condition new proceed-types*

Sets the list of valid proceed types for this condition to *new-proceed-types*.

dbg:special-commands *condition*

Returns a list of all Debugger special commands for this condition.

:proceed-types

This init option defines the set of proceed types to be handled by this instance. *proceed-types* is a list of proceed types (symbols); it must be a subset of the set of proceed types understood by this flavor. If this option is omitted, the instance is able to handle all of the proceed types understood by this flavor in general, but by passing this option explicitly, a subset of acceptable proceed types can be established. This is used by **signal-proceed-case**.

dbg:special-command-p *condition special-command*

Returns **t** if *special-command* is a valid debugger special command for the condition object, *condition*. Returns **nil** otherwise.

dbg:report *condition stream*

Prints the text message associated with this object onto *stream*.

dbg:report-string *condition*	Returns a string containing the report message associated with this object. (sends **dbg:report** to the object.)

22.11.2 Standard Conditions

This section presents the standard condition flavors provided by the system. Some of these flavors are the flavors of actual condition objects that get instantiated in response to certain conditions. Others never actually get instantiated, but are used to build other flavors.

In some cases, the flavor that the system uses to signal an error is not exactly the one listed here, but rather a flavor built on the one listed here. This often comes up when the same error can be reported by different programs that implement a generic protocol. For example, the condition signalled by a remote file-system stream when a file is not found is different from the one signalled by a local file-system stream; however, only the generic **fs:file-not-found** condition should ever be handled by programs, so that a program works regardless of what kind of file-system stream it is using. The exact flavors signalled by each file system are considered to be internal system names, subject to change without notice and not documented herein.

Do not look at system source code to figure out the names of error flavors without being careful to choose the right level of flavor! Furthermore, take care to choose a flavor that can be instantiated if you try to signal a system-defined condition. For example, you can not signal a condition object of type **fs:file-not-found** because this is really a set of errors and this flavor does not handle the **dbg:report** message. If you were to implement your own file system and wanted to signal an error when a file cannot be found, it should probably have its own flavor built on **fs:file-not-found** and other flavors.

Choosing the appropriate condition to handle is a difficult problem. In general you do not want to choose a condition on the basis of the apparent semantics of its name. Rather you should choose it according to the appropriate level of the condition flavor hierarchy. This holds particularly for file-related errors. See the section "File-System Errors", page 617.

There are currently six classes of Standard Conditions:

- Fundamental Conditions
- Lisp Errors
- File-system Errors
- Pathname Errors
- Network Errors
- Tape Errors

Individual classes, their base flavors, and the conditions built on them, are discussed below.

22.11.2.1 Fundamental Conditions

These conditions are basic to the functionality of the condition mechanism, rather than having anything to do with particular system errors.

Here is a summary list of fundamental conditions. More detailed discussion of each follows the listing.

- **condition**
- **dbg:debugger-condition**
- **error**
- **zl:ferror**
- **dbg:proceedable-ferror**
- **sys:no-action-mixin**
- **sys:abort**
- **zl:break**

condition *Flavor*

This is the basic flavor on which all condition objects are built. User-defined conditions are not normally built directly upon **condition**.

dbg:debugger-condition *Flavor*

This flavor is built on **condition**. It is used for entering the Debugger without necessarily classifying the event as an error. This is intended primarily for system use; users should normally build on **error** instead.

error *Flavor*

This flavor is built on **dbg:debugger-condition**. All flavors that represent errors, as opposed to debugger conditions or simple conditions, are built on this flavor.

zl:ferror *Flavor*

This is a simple error flavor for the **zl:ferror** function. Use it when you do not want to invent a new error flavor for a certain condition. Its only state information is an error message, normally created by the call to the **zl:ferror** function. It has two readable and initable instance variables **format-string** and **format-args**. The **zl:format** function is applied to these values to produce the **dbg:report** message.

dbg:proceedable-ferror *Flavor*

This is a simple error flavor for the **zl:fsignal** function. Use it when you do not want to invent a new error flavor for a certain condition, but you want the condition to be proceedable. Its only state information is an error message, created by the call to the **zl:fsignal** function. Its only proceed type is **:no-action**. Proceeding in this way does nothing and causes **zl:fsignal** (or **signal**) to return the symbol **:no-action**.

sys:no-action-mixin *Flavor*

This flavor can be mixed into any condition flavor to define a proceed type called **:no-action**. Proceeding in this way causes the computation to proceed as if no error check had occurred. The signaller might try the action again or might simply go on doing what it would have done. For example, **proceedable-ferror** is just **zl:ferror** with this mixin.

sys:abort *Flavor*

The ABORT key on the keyboard was pressed. This is a simple condition. When **sys:abort** is signalled, control is thrown straight to a restart handler without entering the Debugger.

Note: It is preferable to use (**process-abort *current-process***) instead of (**signal 'sys:abort**) See the section "Special Keys", page 596.

zl:break *Flavor*

This is the flavor of the condition object passed to the Debugger as a result of the m-BREAK command. It is never actually signalled; rather, it is a convention to ensure that the Debugger always has a condition when it assumes control. This is based on **dbg:debugger-condition**. See the section "Special Keys", page 596.

22.11.2.2 Lisp Errors

This section describes the conditions signalled for basic Lisp errors. All of the conditions in this section are based on the **error** flavor unless otherwise indicated.

Lisp errors include the following ten major groups:

- Base Flavor: **sys:cell-contents-error**
- Location Errors
- Base Flavor: **sys:arithmetic-error**
- Base Flavor: **sys:floating-point-exception**
- Miscellaneous System Errors Not Categorized by Base Flavor
- Function-Calling Errors
- Array Errors
- Eval Errors
- Interning Errors Based on **sys:package-error**
- Errors Involving Lisp Printed Representations

22.11.2.3 Base Flavor: sys:cell-contents-error

This group includes the following errors:

- **sys:cell-contents-error**
- **sys:unbound-variable**
- **sys:unbound-symbol**
- **sys:unbound-closure-variable**
- **sys:unbound-instance-variable**
- **sys:undefined-function**
- **sys:bad-data-type-in-memory**

sys:cell-contents-error *Flavor*

All of the kinds of errors resulting from finding invalid contents in a cell of virtual memory are built on this flavor. This represents a set of errors including the various kinds of unbound-variable errors, the undefined-function error, and the bad data-type error.

Proceed type	*Action*
:new-value	Takes one argument, a new value to be used instead of the contents of the cell.
:store-new-value	Takes one argument, a new value to replace the contents of the cell.
:no-action	If you have intervened and stored something into the cell, the contents of the cell can be reread.

sys:unbound-variable *Flavor*

All of the kinds of errors resulting from unbound variables are built on this flavor. Because these are a subset of the "cell contents" errors, this flavor is built on **sys:cell-contents-error**. The **:variable-name** message returns the name of the variable that was unbound (a symbol).

sys:unbound-symbol *Flavor*

An unbound symbol (special variable) was evaluated. Some instances of this flavor provide the **:package-dwim** special command, which takes no arguments and asks whether you want to examine the value of various other symbols with the same print name in other packages. This proceed type is provided only if any such symbols exist in any other packages. (See also **dbg:*defer-package-dwim***.) This flavor is built on **sys:unbound-variable**. The proceed types from **sys:cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:unbound-closure-variable *Flavor*

An unbound closure variable was evaluated. This flavor is built on **sys:unbound-variable**. The proceed types from **cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:unbound-instance-variable *Flavor*

An unbound instance variable was evaluated. The **:instance** message returns the instance in which the unbound variable was found. The proceed types from **cell-contents-error** are provided, as is the :variable-name message from **sys:unbound-variable.**

sys:undefined-function *Flavor*

An undefined function was invoked; that is, an unbound function cell was referenced. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types. The **:function-name** message returns the name of the function that was undefined (a function spec). This also provides **:package-dwim** service, like **sys:unbound-symbol.**

sys:bad-data-type-in-memory *Flavor*

A word with an invalid type code was read from memory. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types.

Message	*Value returned*
:address	virtual address, as a locative pointer, from which the word was read
:data-type	numeric value of the data-type tag field of the word
:pointer	numeric value of the pointer field of the word

22.11.2.4 Location Errors

This group includes the following errors:

- **sys:unknown-setf-reference**
- **sys:unknown-locf-reference**

sys:unknown-setf-reference *Flavor*

zl:setf did not find a **zl:setf** property on the **car** of the form. The **:form** message returns the form that **zl:setf** tried to operate on. This error is signalled when the **zl:setf** macro is expanded.

sys:unknown-locf-reference *Flavor*

locf did not find a **locf** property on the car of the form. The **:form** message returns the form that **locf** tried to operate on. This error is signalled when the **locf** macro is expanded.

22.11.2.5 Base Flavor: sys:arithmetic-error

This group includes the following errors:

- **sys:arithmetic-error**
- **sys:divide-by-zero**
- **sys:non-positive-log**

- **math:singular-matrix**

sys:arithmetic-error *Flavor*

Represents the set of all arithmetic errors. No condition objects of this flavor are actually created; any arithmetic error signals a more specific condition, built on this one. This flavor is provided to make it easy to handle any arithmetic error.

All arithmetic errors handle the **:operands** message. This returns a list of the operands in the operation that caused the error.

sys:divide-by-zero *Flavor*

Division by zero was attempted. This flavor is built on **sys:arithmetic-error**. The **:function** message returns the function that did the division.

sys:non-positive-log *Flavor*

Computation of the logarithm of a nonpositive number was attempted. This flavor is built on **sys:arithmetic-error.** The **:number** message returns the nonpositive number.

math:singular-matrix *Flavor*

A singular matrix was given to a matrix operation such as inversion, taking of the determinant, or computation of the LU decomposition. This flavor is built on **sys:arithmetic-error.**

22.11.2.6 Base Flavor: sys:floating-point-exception

sys:floating-point-exception and the condition flavors based on it are designed to support IEEE floating-point standards. See the section "Numbers", page 83. By default, all IEEE traps are enabled, except for the inexact-result trap. See the special form **without-floating-underflow-traps** in *Symbolics Common Lisp: Language Dictionary.*

The trap handlers that signal these conditions from the system all cause pressing the RESUME key to mean "return the result that would have been returned if the trap had been disabled". For example, pressing RESUME on an overflow returns the appropriately signed infinity as the result. On an underflow it returns the denormalized (possibly zero) result.

This group includes the following errors:

- **sys:floating-point-exception**
- **sys:float-divide-by-zero**
- **sys:floating-exponent-overflow**
- **sys:floating-exponent-underflow**
- **sys:float-inexact-result**

- **sys:float-invalid-operation**
- **sys:float-invalid-compare-operation**
- **sys:negative-sqrt**
- **sys:float-divide-zero-by-zero**

sys:floating-point-exception *Flavor*

This is the base flavor for floating-point exceptional conditions. No condition objects of this flavor are actually created. This flavor is provided to make it easy to handle any floating-point exception. It is built on **sys:arithmetic-error.**

Message	*Value returned*
:operation	A symbol indicating the operation that caused the exception.
:operands	The list of operands to the operation.
:non-trap-result	The result that would have been returned if this trap had been disabled.
:saved-float-operation-status	The value of **sys:float-operation-status** at the time of the exception.

Proceed type	*Action*
:new-value	Takes one argument and uses this value as the result of the operation.

sys:float-divide-by-zero *Flavor*

A floating-point division by zero was attempted. This flavor is built on **sys:divide-by-zero** and **sys:floating-point-exception.**

sys:floating-exponent-overflow *Flavor*

Overflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception.** The **:function** message returns the function that got the overflow, if it is known, and **nil** if it is not known. The **:new-value** proceed type is provided with one argument, a floating-point number to use instead.

sys:floating-exponent-underflow *Flavor*

Underflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception.** The **:function** message returns the function that got the underflow, if it is known, and **nil** if it is not known. The **:use-zero** proceed type is provided with no arguments; a **0.0** is used instead.

sys:float-inexact-result *Flavor*

A floating-point result does not exactly represent the operation's result, due to the fixed precision of floating-point representation. Since most floating-point calculations are inexact, the inexact-result trap is disabled by default. This flavor is built on **sys:floating-point-exception.**

sys:float-invalid-operation *Flavor*

An invalid floating-point operation was attempted, such as dividing infinity by infinity. This flavor is built on **sys:floating-point-exception.**

sys:float-invalid-compare-operation *Flavor*

This is built on and is identical to **sys:float-invalid-operation**, except that it does not expect a numeric result. This flavor is raised for any arithmetic comparison (<, >, ≤, ≥, =, ≠) in which at least one of the operands is a NaN (IEEE not-a-number object).

For example:

```
(< (// 0.0 0.0) 0.0)
```

signals **sys:float-invalid-compare-operation** if you "proceed" from the invalid division by zero operation.

sys:negative-sqrt *Flavor*

Computing the square root of a negative number was attempted. This flavor is built on **sys:float-invalid-operation.**

sys:float-divide-zero-by-zero *Flavor*

A floating-point division of zero by zero was attempted. This flavor is built on **sys:float-invalid-operation** and **sys:float-divide-by-zero.** Most programs handle not this condition itself, but rather one of the component condition flavors.

22.11.2.7 Miscellaneous System Errors Not Categorized by Base Flavor

This group includes the following errors:

- **sys:end-of-file**
- **sys:stream-closed**
- **sys:wrong-stack-group-state**
- **sys:draw-off-end-of-screen**
- **sys:draw-on-unprepared-sheet**
- **sys:bitblt-destination-too-small**
- **sys:bitblt-array-fractional-word-width**
- **sys:write-in-read-only**
- **sys:pdl-overflow**
- **sys:area-overflow**
- **sys:virtual-memory-overflow**
- **sys:region-table-overflow**

- **sys:cons-in-fixed-area**
- **sys:throw-tag-not-seen**
- **sys:instance-variable-zero-referenced**
- **sys:instance-variable-pointer-out-of-range**
- **sys:disk-error**
- **sys:redefinition**

sys:end-of-file *Flavor*

A function doing input from a stream attempted to read past the end-of-file. The **:stream** message returns the stream.

sys:stream-closed *Flavor*

An operation that required a stream to be open was attempted on a closed stream. **sys:stream-closed** accepts the following messages and has corresponding required init keywords:

:attempt	Returns a string briefly describing the attempted action on**stream**, for example, "read from"
:stream	Returns the stream

Example:

```
(error 'sys:stream-closed :attempt "write to" :stream self)
```

sys:wrong-stack-group-state *Flavor*

A stack group was in the wrong state to be resumed. The **:stack-group** message returns the stack group.

sys:draw-off-end-of-screen *Flavor*

Drawing graphics past the edge of the screen was attempted.

sys:draw-on-unprepared-sheet *Flavor*

A drawing primitive (such as **sys:%draw-line**) was used on a screen array not inside a **tv:prepare-sheet** special form. The **:sheet** message returns the sheet (window) that should have been prepared.

sys:bitblt-destination-too-small *Flavor*

The destination array of a **bitblt** was too small.

sys:bitblt-array-fractional-word-width *Flavor*

An array passed to **bitblt** does not have a first dimension that is a multiple of 32 bits. The **:array** message returns the array.

sys:write-in-read-only *Flavor*

Writing into a read-only portion of memory was attempted. The **:address** message returns the address at which the write was attempted.

sys:pdl-overflow *Flavor*

A stack (pdl) overflowed. The **:pdl-name** message returns the name of the stack (a string, such as "regular" or "special"). The **:grow-pdl** proceed type is provided, with no arguments; it increases the size of the stack. This is based on **dbg:debugger-condition**, not on **error**.

sys:area-overflow *Flavor*

This is signalled when the maximum-size (**:size** argument to **make-area**) is exceeded.

sys:virtual-memory-overflow *Flavor*

This is an irrecoverable error that is signalled when you run out of virtual memory.

sys:region-table-overflow *Flavor*

This is an irrecoverable error that is signalled when you run out of regions.

sys:cons-in-fixed-area *Flavor*

Allocation of storage from a fixed area of memory was attempted.

Message	*Value returned*
:area	name of the area
:region	region number

sys:throw-tag-not-seen *Flavor*

throw or **zl:*throw** was called, but no matching **catch** or **zl:*catch** was found.

Message	*Value returned*
:tag	Catch-tag that was being thrown to.
:values	List of the values that were being thrown. If **zl:*throw** was called, this is always a list of two elements, the value being thrown and the tag; if the **throw** special form of Common Lisp is used, the list can be of any length.

The **:new-tag** proceed type is provided with one argument, a new tag (a symbol) to try instead of the original.

sys:instance-variable-zero-referenced *Flavor*

Referencing instance variable 0 of an instance was attempted. This usually means that some method is referring to an instance variable that was deleted by a later evaluation of a **defflavor** form.

sys:instance-variable-pointer-out-of-range *Flavor*

Referencing an instance variable that does not exist was attempted. This usually means that some method is using an obsolete instance because a **defflavor** form got evaluated again and changed the flavor incompatibly.

sys:disk-error *Flavor*

An error was reported by the disk software or controller. The **:retry-disk-operation** proceed type is provided; it takes no arguments.

sys:redefinition *Flavor*

This is a simple condition rather than an error condition. It signals an attempt to redefine something by some other file than the one that originally defined it. The **:definition-type** argument specifies the kind of definition: it might be **defun** if the function cell is being defined, **zl:defstruct** if a structure is being defined, and so on.

Message	*Value returned*
:name	symbol (or function spec) being redefined
:old-pathname	pathname that originally defined it
:new-pathname	pathname that is now trying to define it

Either pathname is **nil** if the definition was from inside the Lisp environment rather than from loading a file.

The following proceed types are provided:

Message	*Action*
:proceed	Redefinition should go ahead; in the future no warnings should be signalled for this pair of pathnames.
:inhibit-definition	Definition is not changed and execution proceeds.
:no-action	Function should be redefined as if no warning had occurred.

Note: if this condition is not handled, the action is controlled by the value of **sys:inhibit-fdefine-warnings**.

22.11.2.8 Function-calling Errors

This group includes the following errors:

- **sys:zero-args-to-select-method**
- **sys:too-few-arguments**
- **sys:too-many-arguments**
- **sys:wrong-type-argument**

sys:zero-args-to-select-method *Flavor*

A select method was applied to no arguments.

sys:too-few-arguments *Flavor*

A function was called with too few arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:additional-arguments** proceed type is provided with one argument, a list of additional argument values to which the function should be applied. If the error is proceeded, these new arguments are appended to the old arguments and the function is called with this new argument list.

sys:too-many-arguments *Flavor*

A function was called with too many arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:fewer-arguments** proceed type is provided with one argument, the new number of arguments with which the function should be called. In proceeding from this error, the function is called with the first *n* arguments only, where *n* is the number specified.

sys:wrong-type-argument *Flavor*

A function was called with at least one argument of invalid type.

Message	*Value returned*
:function	function with invalid argument(s)
:old-value	invalid value
:description	description of valid value
:arg-name	name of the argument
:arg-number	number of the argument (the first argument to a function is **0**, and so on) or **nil** if this does not apply

:description, :arg-name, and **:arg-number** are valid messages only when the error was signalled by **zl:check-arg, zl:check-arg-type**, or **zl:argument-typecase**. Check to be sure that the message is valid before sending it (remember **:operation-handled-p**).

Proceed type	*Action*
:argument-value	Takes one argument, the new value to use for the argument.
:store-argument-value	Takes one argument, the new value to use and to store back into the local variable in which it was found.

22.11.2.9 Array Errors

This group includes the following errors:

- **dbg:bad-array-mixin**
- **sys:bad-array-type**
- **sys:array-has-no-leader**
- **sys:fill-pointer-not-fixnum**
- **sys:array-wrong-number-of-dimensions**
- **sys:array-wrong-number-of-subscripts**
- **sys:number-array-not-allowed**
- **sys:subscript-out-of-bounds**

dbg:bad-array-mixin *Flavor*

Errors involving an array that seems to be the wrong object include this flavor. This condition flavor is never instantiated. It provides the **:array** message, which returns the array.

Proceed type	*Action*
:new-array	Takes one argument, an array to use instead of the old one.
:store-new-array	Takes one argument, an array to use instead of the old one and to store back into the local variable in which it was found.

sys:bad-array-type *Flavor*

A meaningless array type code was found in virtual memory, indicating a system bug. The **:type** message returns the numeric type code.

sys:array-has-no-leader *Flavor*

Using the leader of an array that has no array leader was attempted. The **:array** message returns the array. This includes the **dbg:bad-array-mixin** flavor.

sys:fill-pointer-not-fixnum *Flavor*

The fill pointer of an array was not a fixnum. The **:array** message returns the array. This includes the **dbg:bad-array-mixin** flavor.

sys:array-wrong-number-of-dimensions *Flavor*

The rank of the array provided was wrong; the array is in error and the subscripts are correct.

Message	*Value returned*
:dimensions-given	number of subscripts presented
:dimensions-expected	number that should have been given
:array	the array

This includes the **dbg:bad-array-mixin** flavor.

sys:array-wrong-number-of-subscripts *Flavor*

This assumes that the array is correct and that the user/application caused the error by providing the incorrect number of subscripts.

Message	*Value returned*
:array	The array
:subscripts-given	A list of the subscripts given
:number-of-subscripts-given	The number of subscripts given
:number-of-subscripts-expected	The rank of the array

The following example signals **sys:array-wrong-number-of-subscripts:**

```
(array-in-bounds-p some-3-dimensional-array 2 3)
```

sys:number-array-not-allowed *Flavor*

A number array (such as an **sys:art-4b** or **sys:art-16b**) was used in a context in which number arrays are not valid, such as an attempt to make a pointer to an element with **zl:aloc** or **locf**. This includes the **dbg:bad-array-mixin** flavor.

sys:subscript-out-of-bounds *Flavor*

An attempt was made to reference an array using out-of-bounds subscripts, an out-of-bounds array leader element, or an out-of-bounds instance variable.

Message	*Value returned*
:object	the object (an array or instance) if it is known, and **nil** otherwise
:function	function that did the reference, or **nil** if it is not known
:subscript-used	the subscript that was actually used
:subscript-limit	the limit that it passed

The individual subscripts are reported for the **:subscript-used** and **:subscript-limit** messages. These values are fixnums; if a multidimensional array was used, they are computed products.

Proceed type	*Action*
:new-subscript	Takes an arbitrary number of arguments, the new subscripts for the array reference.
:store-new-subscript	Takes the same arguments as **:new-subscript** and stores them back into the local variables in which they were found.

22.11.2.10 Eval Errors

This group includes the following errors:

- **sys:invalid-function**
- **sys:undefined-keyword-argument**
- **sys:unclaimed-message**

sys:invalid-function *Flavor*

The evaluator attempted to apply an object that is not a function or a symbol whose definition is an object that is not a function. The **:function** message returns the object that was applied. **sys:invalid-function** is signalled but is not proceedable.

sys:undefined-keyword-argument *Flavor*

The evaluator attempted to pass a keyword to a function that does not recognize that keyword.

Message	*Value returned*
:keyword	Unrecognized keyword
:value	The value passed with it

Proceed type	*Action*
:no-action	The keyword and its value are ignored.
:new-keyword	Specifies a new keyword to use instead. Its one argument is the new keyword.

sys:unclaimed-message *Flavor*

This flavor is built on **error**. The flavor system signals this error when it finds a message for which no method is available.

Message	*Value returned*
:object	the object
:message	the message-name
:arguments	the arguments of the message

The object can be an instance or a select method.

22.11.2.11 Interning Errors Based On sys:package-error

This group includes the following errors:

- **sys:package-error**
- **sys:package-not-found**
- **sys:package-locked**

sys:package-error *Flavor*

All package-related error conditions are built on **sys:package-error**.

sys:package-not-found *Flavor*

A package-name lookup did not find any package by the specified name.

The **:name** message returns the name. The **:relative-to** message returns **nil** if only absolute names are being searched, or else the package whose relative names are also searched.

The **:no-action** proceed type can be used to try again. The **:new-name** proceed type can be used to specify a different name or package. The **:create-package** proceed type creates the package with default characteristics.

sys:package-locked *Flavor*

There was an attempt to intern a symbol in a locked package.

The **:symbol** message returns the symbol. The **:package** message returns the package.

The **:no-action** proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

22.11.2.12 Errors Involving Lisp Printed Representations

This group includes the following errors:

- **sys:print-not-readable**
- **sys:parse-error**
- **zl:parse-ferror**
- **sys:read-error**
- **sys:read-premature-end-of-symbol**
- **sys:read-end-of-file**
- **sys:read-list-end-of-file**
- **sys:read-string-end-of-file**

sys:print-not-readable *Flavor*

The Lisp printer encountered an object that it cannot print in a way that the Lisp reader can understand. The printer signals this condition only if **si:print-readably** is not **nil** (it is normally **nil**). The **:object** message returns the object. The **:no-action** proceed type is provided; proceeding this way causes the object to be printed as if **si:print-readably** were **nil**.

sys:parse-error *Flavor*

This flavor is built on **error** and is the type of error caught by the input editor. This flavor accepts the init keyword **:correct-input**. If the value is **t**, which is the default, the input editor prints "Type RUBOUT to correct your input" and does not erase the message until a non-self-inserting character is typed. If the value is **nil**, no message is printed, and any typeout from the read function is erased immediately after the next character is typed. Syntax errors signalled by read functions should be built on top of this flavor.

zl:parse-ferror *Flavor*

This flavor is built on **sys:parse-error** and **zl:ferror**. It accepts the init keywords **:format-string** and **:format-args** as well as **:correct-input**. This flavor exists for read functions that do not have a special flavor of error defined for them.

sys:read-error *Flavor*

This flavor, built on **sys:parse-error**, includes errors encountered by the Lisp reader.

sys:read-premature-end-of-symbol *Flavor*

This is a new error flavor based on **sys:read-error**. It can be used for signalling when some read function finishes reading in the middle of a string that was supposed to contain a single expression.

Message	*Value returned*
:short-symbol	the symbol that was read
:original-string	the string that it was reading from when it finished in the middle

An example of the use of **sys:read-premature-end-of-symbol** is in **zwei:symbol-from-string**.

sys:read-end-of-file *Flavor*

The Lisp reader encountered an end-of-file while in the middle of a string or list. This flavor is built on **sys:read-error** and **sys:end-of-file**.

sys:read-list-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a list. This is built on **sys:read-end-of-file**. The **:list** message returns the list that was being built.

sys:read-string-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a string. This is built on **sys:read-end-of-file**. The **:string** message returns the string that was being built.

22.11.2.13 File-System Errors

The following condition flavors are part of the Symbolics Lisp Machine's generic file system interface. These flavors work for all file systems, whether local Lisp Machine file systems, remote Lisp Machine file systems (accessed over a network), or remote file systems of other kinds, such as UNIX or TOPS-20. All of them report errors uniformly.

Some of these condition flavors describe situations that can occur during any file system operation. These include not only the most basic flavors, such as **fs:file-request-failure** and **fs:data-error**, but also flavors such as **fs:file-not-found** and **fs:directory-not-found**. Other file system condition flavors describe failures related to specific file system operations, such as **fs:rename-failure**, and **fs:delete-failure**. Given all these choices, you have to determine what condition is appropriate to handle, for example in checking for success of a rename operation. Would **fs:rename-failure** include cases where, say, the directory of the file being renamed is not found?

The answer to this question is that you should handle **fs:file-operation-failure**. **fs:rename-failure** and all other conditions at that level are signalled only for errors that relate specifically to the semantics of the operation involved. If you cannot delete a file because the file is not found, **fs:file-not-found** would be signalled. Suppose you cannot delete the file because its "don't delete switch" is set, which is an error relating specifically to deletion. **fs:delete-failure** would be signalled. Therefore, since you cannot know whether a condition flavor related to an operation requested or some more general error is signalled, you usually want to handle one of the most general flavors of file system error.

Under normal conditions, you would bind only for **fs:file-request-failure** or **fs:file-operation-failure** rather than for the more specific condition flavors described in this section. Some guidelines for using the different classes of errors:

error	Any error at all. It is not wise in general to attempt to handle this, because it catches program and operating system bugs as well as file-related bugs, thus "hiding" knowledge of the system problems from you.
fs:file-error	Any file related error at all. This includes **fs:file-operation-failure** as well as **fs:file-request-failure**. Condition objects of flavor **fs:file-request-failure** usually indicate that the file system, host operating system, or network did not operate properly. If your program is attempting to handle file-related errors, it should not handle these: it is usually better to allow the program to enter the debugger. Thus it is very rare that one would want to handle **fs:file-error**.

fs:file-operation-failure	This includes almost all predictable file-related errors, whether they are related to the semantics of a specific operation, or are capable of occurring during many kinds of operations. Therefore, **fs:file-operation-failure** is usually the appropriate condition to handle.
Specific conditions	It is appropriate and correct to handle specific conditions, like **fs:delete-failure**, if your program assigns specific meaning to (or has specific actions associated with) specific occurrences, such as a nonexistent directory or an attempt to delete a protected file. If you do not "care" about specific conditions, but you wish to handle predictable file-related errors, you should handle **fs:file-operation-failure**. You should *not* attempt to handle, say, **fs:delete-failure** to test for any error occurring during deletion; it does not mean that.

File-System errors include the following fifteen major groups:

- **fs:file-error**
- **fs:file-request-failure**
- **fs:file-operation-failure**
- Request Failures Based on **fs:file-request-failure**
- Login Problems
- File Lookup
- **fs:access-error**
- **fs:invalid-pathname-syntax**
- **fs:wrong-kind-of-file**
- **fs:creation-failure**
- **fs:rename-failure**
- **fs:change-property-failure**
- **fs:delete-failure**
- Errors Loading Binary Files
- Miscellaneous Operations Failure

fs:file-error *Flavor*

This set includes errors encountered during file operations. This flavor is built on **error**.

Message	*Value returned*
:pathname	pathname that was being operated on or **nil**
:operation	name of the operation that was being done: this is a keyword symbol such as **:open**, **:close**, **:delete**, or **:change-properties**, and it might be **nil** if the signaller does not know what the operation was or if no specific operation was in progress

In a few cases, the **:retry-file-error** proceed type is provided, with no arguments; it retries the file system request. All flavors in this section accept these messages and might provide this proceed type.

fs:file-request-failure *Flavor*

This set includes all file-system errors in which the request did not manage to get to the file system.

fs:file-operation-failure *Flavor*

This set includes all file-system errors in which the request was delivered to the file system, and the file system decided that it was an error.

Note: every file-system error is either a request failure or an operation failure, and the rules given above explain the distinction. However, these rules are slightly unclear in some cases. If you want to be sure whether a certain error is a request failure or an operation failure, consult the detailed descriptions in the rest of this section.

22.11.2.14 Request Failures Based On fs:file-request-failure

This group includes the following errors:

- **fs:data-error**
- **fs:host-not-available**
- **fs:no-file-system**
- **fs:network-lossage**
- **fs:not-enough-resources**
- **fs:unknown-operation**

fs:data-error *Flavor*

Bad data is in the file system. This might mean data errors detected by hardware or inconsistent data inside the file system. This flavor is built on **fs:file-request-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is provided in some cases; send a **:proceed-types** message to find out.

fs:host-not-available *Flavor*

The file server or file system is intentionally denying service to users. This does *not* mean that the network connection failed; it means that the file system explicitly does not care to be available. This flavor is built on **fs:file-request-failure**.

fs:no-file-system *Flavor*

The file system is not available. For example, this host does not have any file system, or this host's file system cannot be initialized for some reason. This flavor is built on **fs:file-request-failure**.

fs:network-lossage *Flavor*

The file server had some sort of trouble trying to create a new data connection and was unable to do so. This flavor is built on **fs:file-request-failure**.

fs:not-enough-resources *Flavor*

Some resource was not available in sufficient supply. Retrying the operation might work if you wait for some other users to go away or if you close some of your own files. This flavor is built on **fs:file-request-failure**.

fs:unknown-operation *Flavor*

An unsupported file-system operation was attempted. This flavor is built on **fs:file-request-failure**.

22.11.2.15 Login Problems

Some login problems are correctable and some are not. To handle any correctable login problem, you set up a handler for **fs:login-required** rather than handling the individual conditions.

The correctable login problem conditions work in a special way. The Symbolics Lisp Machine's generic file system interface, in the user-end of the remote file protocol, always handles these errors with its own condition handler; it then signals the **fs:login-required** condition. Therefore to handle one of these problems, you set up a handler for **fs:login-required**. The condition object for the correctable login problem can be obtained from the condition object for **fs:login-required** by sending it an **:original-condition** message.

This group includes the following errors:

- **fs:login-problems**
- **fs:correctable-login-problems**
- **fs:unknown-user**
- **fs:invalid-password**
- **fs:not-logged-in**

- **fs:login-required**

fs:login-problems *Flavor*

This set includes all problems encountered while trying to log in to the file system. Currently, none of these ever happen when you use a local file system. This flavor is built on **fs:file-request-failure**.

fs:correctable-login-problems *Flavor*

This set includes all correctable problems encountered while trying to log in to the foreign host. None of these ever happen when you use a local file system. This flavor is built on **fs:login-problems**.

fs:unknown-user *Flavor*

The specified user name is unknown at this host. The **:user-id** message returns the user name that was used. This flavor is built on **fs:correctable-login-problems**.

fs:invalid-password *Flavor*

The specified password was invalid. This flavor is built on **fs:correctable-login-problems**.

fs:not-logged-in *Flavor*

A file operation was attempted before logging in. Normally the file system interface always logs in before doing any operation, but this problem can come up in certain unusual cases in which logging in has been aborted. This flavor is built on **fs:correctable-login-problems**.

fs:login-required *Flavor*

This is a simple condition built on **condition**. It is signalled by the file-system interface whenever one of the correctable login problems happens.

Message	*Value returned*
(send (send error :access-path) :host)	the foreign host
:host-user-id	user name that would be the default for this host
:original-condition	condition object of the correctable login problem

The **:password** proceed type is provided with two arguments, a new user name and a new password, both of which should be strings. If the condition is not handled by any handler, the file system prompts the user for a new user name and password, using the **zl:query-io** stream.

22.11.2.16 File Lookup

This group includes the following errors:

- **fs:file-lookup-error**
- **fs:file-not-found**
- **fs:multiple-file-not-found**
- **fs:directory-not-found**
- **fs:device-not-found**
- **fs:link-target-not-found**

fs:file-lookup-error *Flavor*

This set includes all file-name lookup errors. This flavor is built on **fs:file-operation-failure.**

fs:file-not-found *Flavor*

The file was not found in the containing directory. The TOPS-20 and TENEX "no such file type" and "no such file version" errors also signal this condition. This flavor is built on **fs:file-lookup-error**.

fs:multiple-file-not-found *Flavor*

None of a number of possible files was found. This flavor is built on **fs:file-lookup-error**. It is signalled when **load** is not given a specific file type but cannot find either a source or a binary file to load.

The flavor allows three init keywords of its own. These are also the names of messages that return the following:

:operation	The operation that failed
:pathname	The pathname given to the operation
:pathnames	A list of pathnames that were sought unsuccessfully

The condition has a **:new-pathname** proceed type to prompt for a new pathname.

fs:directory-not-found *Flavor*

The directory of the file was not found or does not exist. This means that the containing directory was not found. If you are trying to open a directory, and the actual directory you are trying to open is not found, **fs:file-not-found** is signalled. This flavor is built on **fs:file-lookup-error**.

This flavor has two Debugger special commands: **:create-directory**, to create only the lowest level of directory, and **:create-directories-recursively**, to create any missing superiors as well.

Errors of this flavor support the **:directory-pathname** message. This message, which can be sent to any such error, returns (when possible) a "pathname as directory" for the actual directory which was not found.

Example:

Assume the directory x:>a>b exists, but has no inferiors. The following produces an error instance to which **:pathname** produces #<LMFS-PATHNAME x:>a>b>c>d>thing.lisp> and **:directory-pathname** produces #<LMFS-PATHNAME x:>a>b>c> >.

```
(open "x:>a>b>c>d>thing.lisp")
```

Note: Not all hosts and access media can provide this information (currently, only local LMFS and LMFS accessed via NFILE can). When a host does not return this information, **:directory-pathname** returns the same as **:pathname**, whose value is a pathname as directory for the best approximation known to the identity of the missing directory.

fs:device-not-found *Flavor*
The device of the file was not found or does not exist. This flavor is built on **fs:file-lookup-error**.

fs:link-target-not-found *Flavor*
The target of the link that was opened did not exist. This flavor is built on **fs:file-lookup-error**.

22.11.2.17 fs:access-error

This group includes the following errors:

- **fs:access-error**
- **fs:incorrect-access-to-file**
- **fs:incorrect-access-to-directory**

fs:access-error *Flavor*
This set includes all protection-violation errors. This flavor is built on **fs:file-operation-failure**.

fs:incorrect-access-to-file *Flavor*
Incorrect access to the file in the directory was attempted. This flavor is built on **fs:access-error**.

fs:incorrect-access-to-directory *Flavor*
Incorrect access to some directory containing the file was attempted. This flavor is built on **fs:access-error**.

22.11.2.18 fs:invalid-pathname-syntax

This group includes the following errors:

- **fs:invalid-pathname-syntax**
- **fs:invalid-wildcard**
- **fs:wildcard-not-allowed**

fs:invalid-pathname-syntax *Flavor*

This set includes all invalid pathname syntax errors. This is not the same as **fs:parse-pathname-error.** (See the flavor **fs:parse-pathname-error,** page 629.) These errors occur when a successfully parsed pathname object is given to the file system, but something is wrong with it. See the specific conditions that follow. This flavor is built on **fs:file-operation-failure.**

fs:invalid-wildcard *Flavor*

The pathname is not a valid wildcard pathname. This flavor is built on **fs:invalid-pathname-syntax.**

fs:wildcard-not-allowed *Flavor*

A wildcard pathname was presented in a context that does not allow wildcards. This flavor is built on **fs:invalid-pathname-syntax.**

22.11.2.19 fs:wrong-kind-of-file

This group includes the following errors:

- **fs:wrong-kind-of-file**
- **fs:invalid-operation-for-link**
- **fs:invalid-operation-for-directory**

fs:wrong-kind-of-file *Flavor*

This set includes errors in which an invalid operation for a file, directory, or link was attempted.

fs:invalid-operation-for-link *Flavor*

The specified operation is invalid for links, and this pathname is the name of a link. This flavor is built on **fs:wrong-kind-of-file.**

fs:invalid-operation-for-directory *Flavor*

The specified operation is invalid for directories, and this pathname is the name of a directory. This flavor is built on **fs:wrong-kind-of-file.**

22.11.2.20 fs:creation-failure

This group includes the following errors:

- **fs:creation-failure**
- **fs:file-already-exists**
- **fs:create-directory-failure**
- **fs:directory-already-exists**
- **fs:create-link-failure**

fs:creation-failure *Flavor*

This set includes errors related to attempts to create a file, directory, or link. This flavor is built on **fs:file-operation-failure.**

fs:file-already-exists *Flavor*

A file of this name already exists. This flavor is built on **fs:creation-failure.**

fs:create-directory-failure *Flavor*

This set includes errors related to attempts to create a directory. This flavor is built on **fs:creation-failure.**

fs:directory-already-exists *Flavor*

A directory or file of this name already exists. This flavor is built on **fs:creation-directory-failure.**

fs:create-link-failure *Flavor*

This set includes errors related to attempts to create a link. This flavor is built on **fs:creation-failure.**

22.11.2.21 fs:rename-failure

This group includes the following errors:

- **fs:rename-failure**
- **fs:rename-to-existing-file**
- **fs:rename-across-directories**
- **fs:rename-across-hosts**

fs:rename-failure *Flavor*

This set includes errors related to attempts to rename a file. The **:new-pathname** message returns the target pathname of the rename operation. This flavor is built on **fs:file-operation-failure.**

fs:rename-to-existing-file *Flavor*

The target name of a rename operation is the name of a file that already exists. This flavor is built on **fs:rename-failure**.

fs:rename-across-directories *Flavor*

The devices or directories of the initial and target pathnames are not the same, but on this file system they are required to be. This flavor is built on **fs:rename-failure**.

fs:rename-across-hosts *Flavor*

The hosts of the initial and target pathnames are not the same. This flavor is built on **fs:rename-failure**.

22.11.2.22 fs:change-property-failure

This group includes the following errors:

- **fs:change-property-failure**
- **fs:unknown-property**
- **fs:invalid-property-value**

fs:change-property-failure *Flavor*

This set includes errors related to attempts to change properties of a file. This might mean that you tried to set a property that only the file system is allowed to set. For file systems without user-defined properties, it might mean that no such property exists. This flavor is built on **fs:file-operation-failure**.

fs:unknown-property *Flavor*

The property is unknown. This flavor is built on **fs:change-property-failure**.

fs:invalid-property-value *Flavor*

The new value provided for the property is invalid. This flavor is built on **fs:change-property-failure**.

22.11.2.23 fs:delete-failure

This group includes the following errors:

- **fs:delete-failure**
- **fs:directory-not-empty**
- **fs:dont-delete-flag-set**

fs:delete-failure *Flavor*

This set includes errors related to attempts to delete a file. It applies to cases where the file server reports that it cannot delete a file. The exact events involved depend on what the host file server has received from the host. This flavor is built on **fs:file-operation-failure.**

fs:directory-not-empty *Flavor*

An invalid deletion of a nonempty directory was attempted. This flavor is built on **fs:delete-failure.**

fs:dont-delete-flag-set *Flavor*

Deleting a file with a "don't delete" flag was attempted. This flavor is built on **fs:delete-failure.**

22.11.2.24 Miscellaneous Operations Failures

This group includes the following errors:

- **fs:circular-link**
- **fs:unimplemented-option**
- **fs:inconsistent-options**
- **fs:invalid-byte-size**
- **fs:no-more-room**
- **fs:filepos-out-of-range**
- **fs:file-locked**
- **fs:file-open-for-output**
- **fs:not-available**

fs:circular-link *Flavor*

The pathname is a link that eventually gets linked back to itself. This flavor is built on **fs:file-operation-failure.**

fs:unimplemented-option *Flavor*

This set includes errors in which an option to a command is not implemented. This flavor is built on **fs:file-operation-failure.**

fs:inconsistent-options *Flavor*

Some of the options given in this operation are inconsistent with others. This flavor is built on **fs:file-operation-failure.**

fs:invalid-byte-size *Flavor*

The value of the "byte size" option was not valid. This flavor is built on **fs:unimplemented-option.**

fs:no-more-room *Flavor*

The file system is out of room. This can mean any of several things:

- The entire file system might be full
- The particular volume that you are using might be full
- Your directory might be full
- You might have run out of your allocated quota
- Other system-dependent things

This flavor is built on **fs:file-operation-failure.** The **:retry-file-operation** proceed type from **fs:file-error** is sometimes provided.

fs:filepos-out-of-range *Flavor*

Setting the file pointer past the end-of-file position or to a negative position was attempted. This flavor is built on **fs:file-operation-failure.**

fs:file-locked *Flavor*

The file is locked. It cannot be accessed, possibly because it is in use by some other process. Different file systems can have this problem in various system-dependent ways. This flavor is built on **fs:file-operation-failure.**

fs:file-open-for-output *Flavor*

Opening a file that was already opened for output was attempted. This flavor is built on **fs:file-operation-failure.** Note: ITS, TOPS-20, and TENEX file servers do not use this condition; they signal **fs:file-locked** instead.

fs:not-available *Flavor*

The file or device exists but is not available. Typically, the disk pack is not mounted on a drive, the drive is broken, or the like. Probably operator intervention is required to fix the problem, but retrying the operation is likely to work after the problem is solved. This flavor is built on **fs:file-operation-failure.** Do not confuse this with **fs:host-not-available.**

22.11.2.25 Errors Loading Binary Files

This group includes the following errors:

- **sys:binary-file-obsolete-version**
- **sys:binary-file-obsolete-version-3**

sys:binary-file-obsolete-version *Flavor*

A condition based on **sys:binary-file-obsolete-version** is signalled whenever the system reads a binary file whose version is obsolete but that can still be loaded (with possible incorrect results). If you want the file loaded, proceed from this condition with **:no-action.**

sys:binary-file-obsolete-version-3 *Flavor*

A condition based on **sys:binary-file-obsolete-version-3** is signalled whenever the system reads a version 3 (Release 6) binary file. The file can still be loaded with possible incorrect results. If you want the file loaded, proceed from this condition with **:no-action**.

22.11.2.26 Pathname Errors

Pathname errors have the following five major groups:

- **fs:pathname-error**
- **fs:parse-pathname-error**
- **fs:invalid-pathname-component**
- **fs:unknown-pathname-host**
- **fs:undefined-logical-pathname-translation**

fs:pathname-error *Flavor*

This set includes errors related to pathnames. This is built on the **error** flavor. The following flavors are built on this one.

fs:parse-pathname-error *Flavor*

A problem occurred in attempting to parse a pathname.

fs:invalid-pathname-component *Flavor*

Attempt to create a pathname with an invalid component.

Message	*Value returned*
:pathname	the pathname
:component-value	the invalid value
:component	the name of the component (a keyword symbol such as **:name** or **:directory**)
:component-description	a "pretty name" for the component (such as file name or directory)

The **:new-component** proceed type is defined with one argument, a component value to use instead.

At the time this is signalled, a pathname object with the invalid component has actually been created; this is what the **:pathname** message returns. The error is signalled just after the pathname object is created before it goes in the pathname hash table.

fs:unknown-pathname-host *Flavor*

The function **fs:get-pathname-host** was given a name that is not the name of any known file computer. The **:name** message returns the name (a string).

fs:undefined-logical-pathname-translation *Flavor*

A logical pathname was referenced but is not defined. The **:logical-pathname** message returns the logical pathname. This flavor has a **:define-directory** proceed type, which prompts for a physical pathname whose directory component is the translation of the logical directory on the given host.

22.11.2.27 Network Errors

Network errors have the following four major groups:

- **sys:network-error**
- Local Network Problems
- Remote Network Problems
- Connection Problems

sys:network-error *Flavor*

This set includes errors signalled by networks. These are generic network errors that are used uniformly for any supported networks. This flavor is built on **error.**

22.11.2.28 Local Network Problems

This group includes the following errors:

- **sys:local-network-error**
- **sys:network-resources-exhausted**
- **sys:unknown-address**
- **sys:unknown-host-name**

sys:local-network-error *Flavor*

This set includes network errors related to problems with one's own Symbolics Lisp Machine rather than with the network or the foreign host. This flavor is built on **sys:network-error.**

sys:network-resources-exhausted *Flavor*

The local network control program exhausted some resource; for example, its connection table is full. This flavor is built on **sys:local-network-error.**

sys:unknown-address *Flavor*

The network control program was given an address that is not understood. The **:address** message returns the address. This flavor is built on **sys:local-network-error.**

sys:unknown-host-name *Flavor*

The host parser (**net:parse-host**) was given a name that is not the name of any known host. The **:name** message returns the name. This flavor is built on **sys:local-network-error.**

22.11.2.29 Remote Network Problems

This group includes the following errors:

- **sys:remote-network-error**
- **sys:bad-connection-state**
- **sys:connection-error**
- **sys:host-not-responding**

sys:remote-network-error *Flavor*

This set includes network errors related to problems with the network or the foreign host, rather than with one's own Symbolics Lisp Machine.

Message	*Value returned*
:foreign-host	the remote host
:connection	the connection or **nil** if no particular connection is involved

This flavor is built on **sys:network-error.**

sys:bad-connection-state *Flavor*

This set includes remote errors in which a connection enters a bad state. This flavor is built on **sys:remote-network-error**. It actually can happen due to local causes, however. In particular, if your Symbolics Lisp Machine stays inside a **without-interrupts** for a long time, the network control program might decide that a host is not answering periodic status requests and put its connections into a closed state.

sys:connection-error *Flavor*

This set includes remote errors that occur while trying to establish a new network connection. The **:contact-name** message to any error object in this set returns the contact name that you were trying to connect to. This flavor is built on **sys:remote-network-error.**

sys:host-not-responding *Flavor*

This set includes errors in which the host is not responding, whether during initial connection or in the middle of a connection. This flavor is built on **sys:remote-network-error.**

22.11.2.30 Connection Problems

This group includes the following errors:

- **sys:host-not-responding-during-connection**
- **sys:host-stopped-responding**
- **sys:connection-refused**
- **sys:connection-closed**
- **sys:connection-closed-locally**
- **sys:connection-lost**
- **sys:connection-no-more-data**
- **sys:network-stream-closed**

sys:host-not-responding-during-connection *Flavor*

The network control program timed out while trying to establish a new connection to a host. The host might be down, or the network might be down. This flavor is built on **sys:host-not-responding** and **sys:connection-error.**

sys:host-stopped-responding *Flavor*

A host stopped responding during an established network connection. The host or the network might have crashed. This flavor is built on **sys:host-not-responding** and **sys:bad-connection-state.**

sys:connection-refused *Flavor*

The foreign host explicitly refused to accept the connection. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:connection-error.**

sys:connection-closed *Flavor*

An established connection became closed. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state.**

sys:connection-closed-locally *Flavor*

The local host closed the connection and then tried to use it. This flavor is built on **sys:bad-connection-state.**

sys:connection-lost *Flavor*

The foreign host reported a problem with an established connection and that connection can no longer be used. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state.**

sys:connection-no-more-data *Flavor*

No more data remains on this connection. This flavor is built on **sys:bad-connection-state.**

sys:network-stream-closed *Flavor*

This is a combination of **sys:network-error** and **sys:stream-closed** and is usually used as a base flavor by network implementations (for example, Chaos and TCP).

22.11.2.31 Tape Errors

Tape errors have the following five major groups:

- **tape:tape-error**
- **tape:mount-error**
- **tape:tape-device-error**
- **tape:end-of-tape**

tape:tape-error *Flavor*

This set includes all tape errors. This flavor is built on **error.**

tape:mount-error *Flavor*

A set of errors signalled because a tape could not be mounted. This includes problems such as no ring and drive not ready. Normally, **tape:make-stream** handles these errors and manages mount retry. This flavor is built on **tape:tape-error.**

tape:tape-device-error *Flavor*

A hardware data error, such as a parity error, controller error, or interface error, occurred. This flavor has **tape:tape-error** as a **:required-flavor.**

tape:end-of-tape *Flavor*

The end of the tape was encountered. When this happens on writing, the tape usually has a few more feet left, in which the program is expected to finish up and write two end-of-file marks. Normally, closing the stream does this automatically. Whether or not this error is ever seen on input depends on the tape controller. Most systems do not see the end of tape on reading, and rely on the software that wrote the tape to have cleanly terminated its data, with EOFs.

This flavor is built on **tape:tape-device-error** and **tape:tape-error.**

23. Packages

23.1 The Need for Packages

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

Genera is a huge Lisp environment, in which many programs must coexist. All of the "operating system", the compiler, the editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program that you use during your session must be loaded into the same environment. Each of these programs is composed of a group of functions; each function must have its own distinct name to avoid conflicts. For example, if the compiler had a function named **pull**, and you loaded a program that had its own function named **pull**, the compiler's **pull** would be redefined, probably breaking the compiler.

It would not really be possible to prevent these conflicts, since the programs are written by many different people who could never get together to hash out who gets the privilege of using a specific name such as **pull**.

Now, if two programs are to coexist in the Lisp world, each with its own function **pull**, then each program must have its own symbol named "**pull**", because one symbol cannot have two function definitions. The same reasoning applies to any other use of symbols to name things. Not only functions but variables, flavors, and many other things are named with symbols, and hence require isolation between symbols belonging to different programs.

A *package* is a mapping from names to symbols. When two programs are not closely related and hence are likely to have conflicts over the use of names, the programs can use separate packages to enable each program to have a different mapping from names to symbols. In the example above, the compiler can use a package that maps the name **pull** into a symbol whose function definition is the compiler's **pull** function. Your program can use a different package that maps the name **pull** into a different symbol whose function definition is your function. When your program is loaded, the compiler's **pull** function is not redefined, because it is attached to a symbol that is not affected by your program. The compiler does not break.

The word "package" is used to refer to a mapping from names to symbols because a number of related symbols are packaged together into a single entity. Since the substance of a program (such as its function definitions and variables) consists of attributes of symbols, a package also packages together the parts of a program. The package system allows the author of a group of closely related programs that should share the same symbols to define a single package for those programs.

It is important to understand the distinction between a name and a symbol. A name is a sequence of characters that appears on paper (or on a screen or in a file). This is often called a *printed representation.* A symbol is a Lisp object inside the machine. You should keep in mind how Lisp reading and loading work. When a source file is read into Genera, or a compiled binary file is loaded in, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it uses a package to map that printed representation (a name) into the symbol itself. The loader does the same thing. The package system arranges to use the correct package whenever a file is read or loaded. (For a detailed explanation of this process: See the section "Specifying Packages in Programs", page 643.

23.1.0.1 Example of the Need for Packages

Suppose there are two programs named **chaos** and **arpa**, for handling the Chaosnet and Arpanet respectively. The author of each program wants to have a function called **get-packet**, which reads in a packet from the network. Also, each wants to have a function called **allocate-pbuf**, which allocates the packet buffer. Each "get" routine first allocates a packet buffer, and then reads bits into the buffer; therefore, each version of **get-packet** should call the respective version of **allocate-pbuf**.

Without the package system, the two programs could not coexist in the same Lisp environment. But the package system can be used to provide a separate space of names for each program. What is required is to define a package named **chaos** to contain the Chaosnet program, and another package **arpa** to hold the Arpanet program. When the Chaosnet program is read into the machine, the names it uses are translated into symbols via the **chaos** package. So when the Chaosnet program's **get-packet** refers to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** package is found, which is the **allocate-pbuf** of the Chaosnet program – the right one. Similarly, the Arpanet program's **get-packet** would be read in using the **arpa** package and would refer to the Arpanet program's **allocate-pbuf**.

23.2 Sharing of Symbols Among Packages

23.2.0.1 How the Package System Allows Symbol Sharing

Besides keeping programs isolated by giving each program its own set of symbols, the package system must also provide controlled sharing of symbols among packages. It would not be adequate for each package's set of symbols to be completely disjoint from the symbols of every other package. For example, almost every package ought to include the whole Lisp language: **car**, **cdr**, **zl:format**, and so on should be available to every program.

There is a critical tension between these two goals of the package system. On the one hand, we want to keep the packages isolated, to avoid the need to think about conflicts between programs when we choose names for things. On the other hand, we want to provide connections among packages so that the facilities of one program can be made available to other programs. All the complexity of the package system arises from this tension. Almost all of the package system's features exist to provide easy ways to control the sharing of symbols among packages, while avoiding accidental unwanted sharing of symbols. Unexpected sharing of a symbol between packages, when the authors of the programs in those packages expected to have private symbols of their own, is a *name conflict* and can cause programs to go awry. See the section "Package Name-Conflict Errors", page 654.

Note that sharing symbols is not as simple as merely making the symbols defined by the Lisp language available in every package. A very important feature of Genera is *shared programs*; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with many other systems, where many different programs have been written to accomplish the same thing.

For example, the routines to manipulate a robot arm might be a separate program, residing in its own package. A second program called **blocks** (the blocks world, of course) wants to manipulate the arm, so it would want to call functions from the arm package. This means that the blocks program must have a way to name those robot arm functions. One way to do this is to arrange for the name-to-symbol mapping of the blocks package to map the names of those functions into the same identical symbols as the name-to-symbol mapping of the arm package. These symbols would then be shared between the two packages.

This sharing must be done with great care. The symbols to be shared between the two packages constitute an interface between two modules. The names to be shared must be agreed upon by the authors of both programs, or at least known to them. They cannot simply make every symbol in the arm program available to the blocks program. Instead they must define some subset of the symbols used by the arm program as its *interface* and make only those symbols available. Typically each name in the interface is carefully chosen (more carefully than names that are only used internally). The arm program comes with documentation listing the symbols that constitute its interface and describing what each is used for. This tells the author of the blocks program not only that a particular symbol is being used as the name of a function in the arms program (and thus cannot be used for a function elsewhere), but also what that function does (move the arm, for instance) when it is called.

The package system provides for several styles of interface between modules. For several examples of how the blocks program and the arm program might communicate: See the section "Examples of Symbol Sharing Among Packages", page 651.

An important aspect of the package system is that it makes it necessary to clarify the modularity of programs and the interfaces between them. The package system provides some tools to allow the interface to be explicitly defined and to check that everyone agrees on the interface.

23.2.1 External Symbols

The name-to-symbol mappings of a package are divided into two classes, *external* and *internal.* We refer to the symbols accessible via these mappings as being *external* and *internal* symbols of the package in question, though really it is the mappings that are different and not the symbols themselves. Within a given package, a name refers to one symbol or to none; if it does refer to a symbol, that symbol is either external or internal in that package, but not both.

External symbols are part of the package's public interface to other packages. These are supposed to be chosen with some care and are advertised to outside users of the package. Internal symbols are for internal use only, and these symbols are normally hidden from other packages. Most symbols are created as internal symbols; they become external only if they are explicitly *exported* from a package.

A symbol can appear in many packages. It can be external in one package and internal in another. It is valid for a symbol to be internal in more than one package, and for a symbol to be external in more than one package. A name can refer to different symbols in different packages. However, a symbol always has the same name no matter where it appears. This restriction is imposed both for conceptual simplicity and for ease of implementation.

23.2.2 Package Inheritance

Some name-to-symbol mappings are established by the package itself, while others are inherited from other packages. When package A inherits mappings from package B, package A is said to *use* package B. A symbol is said to be *accessible* in a package if its name maps to it in that package, whether directly or by inheritance. A symbol is said to be *present* in a package if its name maps to it directly (not by inheritance). If a symbol is accessible to a package, then it can be referenced by a program that is read into that package. Inheritance allows a package to be built up by combining symbols from a number of other packages.

Package inheritance interacts with the distinction between internal and external symbols. When one package uses another, it inherits only the external symbols of that package. This is necessary in order to provide a well-defined interface and avoid accidental name conflicts. The external symbols are the ones that are carefully chosen and advertised. If internal symbols were inherited, it would be hard to predict just which symbols were shared between packages.

A package can use any number of other packages; it inherits the external symbols of all of them. If two of these external symbols had the same name it would be unpredictable which one would be inherited, so this is considered to be a name-conflict error. Consequently the order of the used packages is immaterial and does not affect what symbols are accessible.

Only symbols that are present in a package can be external symbols of that package. However, the package system hides this restriction by copying an inherited mapping directly into a package if you request that the symbol be exported. Note: When package A uses package B, it inherits the external symbols of B. But these do not become external symbols of A, and are not inherited by package C that uses package A. A symbol becomes an external symbol of A only by an explicit request to export it from A.

A package can be made to use another package by the **:use** option to **defpackage** or **make-package** or by calling the **use-package** function.

23.2.3 Global Packages

Almost every package should have the basic symbols of the Lisp language accessible to it. This includes:

- Symbols that are names of useful functions, such as **cdr**, **cons**, and **print**
- Symbols that are names of special forms, such as **cond**
- Symbols that are names of useful variables, such as ***read-base***, ***standard-output***, and *****
- Symbols that are names of useful constants, such as **lambda-list-keywords**
- Symbols that are used by the language as symbols in their own right, such as **&optional**, **t**, **nil**, and **special**

Rather than providing an explicit interface between every program and the Lisp language, listing explicitly the particular symbols from the Lisp language that that program intends to use, it is more convenient to make all the Lisp symbols accessible. Unless otherwise specified, every package inherits from a global package. Common Lisp packages inherit from **common-lisp-global** (or **cl**) and Zetalisp packages inherit from **global** (or **zl**). The external symbols of **common-lisp-global** are all the symbols of the Lisp language, including all the symbols documented without a colon (:) in their name. The **common-lisp-global** package has no internal symbols.

All programs share the global symbols, and cannot use them for private purposes. For example, the symbol **delete** is the name of a Lisp function and thus is in the **common-lisp-global** package. Even if a program does not use the **delete** function, it inherits the global symbol named **delete** and therefore cannot define its own function with that name to do something different. Furthermore, if two programs each want to use the symbol **delete** as a property list indicator, they can bump into each other because they do not have private symbols. You can use a mechanism called *shadowing* to declare that a private symbol is desired rather than inheriting the global symbol. See the section "Shadowing Symbols", page 642. You can also use the **where-is** function and the Where Is Symbol (m-X) editor command to determine whether a symbol is private or shared when writing a program.

Similar to the **common-lisp-global** package is the **system** package, which contains all the symbols that are part of the "operating system" interface or the machine architecture, but not regarded as part of the Lisp language. The **system** package is not inherited unless specifically requested.

Here is how package inheritance works in the example of the two network programs. (See the section "Example of the Need for Packages", page 636.) When the Chaosnet program is read into the Lisp world, the current package is the **chaos** package. Thus all of the names in the Chaosnet program are mapped into symbols by the **chaos** package. If there is a reference to some well-known global symbol such as **append**, it is found by inheritance from the **common-lisp-global** package, assuming no symbol by that name is present in the **chaos** package. If, however, there is a reference to a symbol that you created, a new symbol is created in the **chaos** package. Suppose the name **get-packet** is referenced for the first time. No symbol by this name is directly present in the **chaos** package, nor is such a symbol inherited from **common-lisp-global**. Therefore the reader (actually the **intern** function) creates a new symbol named **get-packet** and makes it present in the **chaos** package. When **get-packet** is referred to later in the Chaosnet program, that symbol is found.

When the Arpanet program is read in, the current package is **arpa** instead of **chaos**. When the Arpanet program refers to **append**, it gets the **common-lisp-global** one; that is, it shares the same symbol that the Chaosnet program got. However, if it refers to **get-packet**, it does *not* get the same symbol the Chaosnet program got, because the **chaos** package is not being searched. Rather, the **arpa** and **common-lisp-global** packages are searched. A new symbol named **get-packet** is created and made present in the **arpa** package.

So what has happened is that there are two **get-packets**: one for **chaos** and one for **arpa**. The two programs are loaded together without name conflicts.

23.2.4 Home Package of a Symbol

Every symbol has a *home package*. When a new symbol is created by the reader and made present in the current package, its home package is set to the current package. The home package of a symbol can be obtained with the **symbol-package** function.

Most symbols are present only in their home package; however, it is possible to make a symbol be present in any number of packages. Only one of those packages can be distinguished as the home package; normally this is the first package in which the symbol was present. The package system tries to ensure that a symbol *is* present in its home package. When a symbol is first created by the reader (actually by the **intern** function), it is guaranteed to be present in its home package. If the symbol is removed from its home package (by the **unintern** function), the home package of the symbol is set to **nil**, even if the symbol is still present in some other package.

Some symbols are not present in any package; they are said to be *uninterned.* See the section "Mapping Names to Symbols", page 659. The **make-symbol** function can be used to create such a symbol. An uninterned symbol has no home package; the **symbol-package** function returns **nil** given such a symbol.

When a symbol is printed, for example, with **prin1**, the printer produces a printed representation that the reader turns back into the same symbol. If the symbol is not accessible to the current package, a qualified name is printed. See the section "Qualified Package Names", page 648. The symbol's home package is used as the prefix in the qualified name.

23.2.5 Importing and Exporting Symbols

A symbol can be made accessible to packages other than its home package in two ways, *importing* and *exporting.*

Any symbol can be made present in a package by *importing* it into that package. This is how a symbol can be present in more than one package at the same time. After importing a symbol into the current package, it can be referred to directly with an unqualified name. Importing a symbol does not change its home package, and does not change its status in any other packages in which it is present.

When a symbol is imported, if another symbol with the same name is already accessible to the package, a name-conflict error is signalled. The *shadowing-import* operation is a combination of shadowing (See the section "Shadowing Symbols", page 642.) and importing; it resolves a name conflict by getting rid of any existing symbol accessible to the package.

Any number of symbols can be *exported* from a package. This declares them to be *external* symbols of that package and makes them accessible in any other packages that *use* the first package. To use a package means to inherit its external symbols.

When a symbol is exported, the package system makes sure that no name conflict is caused in any of the packages that inherit the newly exported symbol.

A symbol can be imported by using the **:import**, **:import-from**, or **:shadowing-import** option to **defpackage** and **make-package**, or by calling the

import or **shadowing-import** function. A symbol can be exported by using the **:export** option to **defpackage** or **make-package**, or by calling the **export** function. See the section "Defining a Package", page 658. See the section "Functions That Import, Export, and Shadow Symbols", page 662.

23.2.6 Shadowing Symbols

You can avoid inheriting unwanted symbols by *shadowing* them. To shadow a symbol that would otherwise be inherited, you create a new symbol with the same name and make it present in the package. The new symbol is put on the package's list of shadowing symbols, to tell the package system that it is not an accident that there are two symbols with the same name. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. Shadowing allows the creator of a package to avoid name conflicts that are anticipated in advance.

As an example of shadowing, suppose you want to define a function named **nth** that is different from the normal **nth** function. (Perhaps you want **nth** to be compatible with the Interlisp function of that name.) Simply writing **(defun nth ...)** in your program would redefine the system-provided **nth** function, probably breaking other programs that use it. (The system detects this and queries you before proceeding with the redefinition.)

The way to resolve this conflict is to put the program (call it snail) that needs the incompatible definition of **nth** in its own package and to make the **snail** package shadow the symbol **nth**.

Now there are two symbols named **nth**, so defining **snail's nth** to be an Interlisp-compatible function does not affect the definition of the global **nth**. Inside the snail program, the global symbol **nth** cannot be seen, which is why we say that it is shadowed. If some reason arises to refer to the global symbol **nth** inside the snail program, the qualified name **global:nth** can be used.

A shadowing symbol can be established by the **:shadow** or **:shadowing-import** option to **defpackage** or **make-package**, or by calling the **shadow** or **shadowing-import** function. See the section "Functions That Import, Export, and Shadow Symbols", page 662.

23.2.6.1 The Keyword Package

The Lisp reader is not context-sensitive; it reads the same printed representation as the same symbol regardless of whether the symbol is being used as the name of a function, the name of a variable, a quoted constant, a syntactic word in a special form, or anything else. The consistency and simplicity afforded by this lack of context sensitivity are very important to Lisp's interchangeability of programs and data, but they do cause a problem in connection with packages. If a certain function is to be shared between two packages, then the symbol that names that function has to be shared for all contexts, not just for functional context. This can accidentally cause a variable, or a property list indicator, or some other use of a symbol, to be shared between two packages when not desired. Consequently, it is important to minimize the number of symbols that are shared between packages, since every such symbol becomes a "reserved word" that cannot be used without thinking about the implications. Furthermore, the set of symbols shared among all the packages in the world is not legitimately user-extensible, because adding a new shared symbol could cause a name conflict between unrelated programs that use symbols by that name for their own private purposes.

On the other hand, there are many important applications for which the package system just gets in the way and one would really like to have *all* symbols shared between packages. Typically this occurs when symbols are used as objects in their own right, rather than just as names for things.

This dilemma is partially resolved by the introduction of *keywords* into the language. Keywords are a set of symbols that is disjoint from all other symbols and exist as a completely independent set of names. There is no separation of packages as far as keywords are concerned; all keywords are available to all packages and two distinct keywords cannot have the same name. Of course, a keyword can have the same name as one or more ordinary symbols. To distinguish keywords from ordinary symbols, the printed representation of a keyword starts with a colon (:) character. The sharing of keywords among all packages does not affect the separation of ordinary symbols into private symbols of each package.

23.3 Specifying Packages in Programs

If you are an inexperienced user, you need never be aware of the existence of packages when writing programs. The **user** package is selected by default as the package for reading expressions typed at the Lisp Listener. Files are read in the **user** package if no package is specified. Since all the functions that users are likely to need are provided in the **global** package, which is used by **user**, they are all accessible. In the documentation, functions that are not in the **global** package are documented with colons in their names, so typing the name the way it is documented works. Keywords, of course, must be typed with a prefix colon, but since that is the way they are documented it is possible to regard the colon as just part of the name, not as anything having to do with packages.

The current package is the value of the variable ***package*** (**zl:package**). The current package in the "selected" process is displayed in the status line. This allows you to tell how forms you type in are read.

If you are writing a program that you expect others to use, you should put it in some package other than **user**, so that its internal functions do not conflict with names other users use. For whatever reason, if you are loading your programs into packages other than **user**, you need to know about special constructs including **defpackage**, qualified names, and file attribute lists. See the section "Defining a Package", page 658. See the section "Qualified Package Names", page 648.

Obviously, every file must be loaded into the right package to serve its purpose. It might not be so obvious that every file must be compiled in the right package, but it is just as true. Any time the names of symbols appearing in the file must be converted to the actual symbols, the conversion must take place relative to a package.

The system usually decides which package to use for a file by looking at the file's *attribute list*. See the section "File Attribute Lists" in *Reference Guide to Streams, Files, and I/O*. A compiled file remembers the name of the package it was compiled in, and loads into the same package. In the absence of any of these specifications, the package defaults to the current value of ***package***, which is usually the **user** package unless you change it.

The file attribute list of a character file is the line at the front of the file that looks something like:

```
;;; -*- Mode:Lisp; Package:System-Internals -*-
```

This specifies that the package whose name or nickname is **system-internals** is to be used. Alphabetic case does not matter in these specifications. Relative package names are not used, since there is no meaningful package to which the name could be relative. See the section "Relative Package Names", page 646.

If the package attribute contains parentheses, then the package is automatically created if it is not found. This is useful when a single file is in its own package, not shared with any other files, and no special options are required to set up that package. The valid forms of package attribute are:

-*- Package: *Name* **-*-**
: Signal an error if the package is not found, allowing you to load the package's definition from another file, specify the name of an existing package to use instead, or create the package with default characteristics.

-*- Package: (*Name***) -*-**
: If the package is not found, create it with the specified name and default characteristics. It uses **global** so that it inherits the Lisp language symbols.

-*- Package: (*Name use***) -*-**
: If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages.

-*- Package: (*Name use size*) -*-

If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages. *size* is a decimal number, the number of symbols that expected to be present in the package.

-*- Package: (*Name keyword value keyword value...*) -*-

If the package is not found, create it with the specified name. The rest of the list supplies the keyword arguments to **make-package**. In the event of an ambiguity between this form and the previous one, the previous one is preferred. You can avoid ambiguity by specifying more than one keyword.

Binary files have similar file attribute lists. The compiler always puts in a **:package** attribute to cause the binary file to be loaded into the same package it was compiled in, unless this attribute is overridden by arguments to **load**.

23.4 Package Names

23.4.1 Introduction to Package Names

Each package has a name and perhaps some nicknames. These are assigned when the package is created, though they can be changed later. A package's name should be something long and self-explanatory like **editor**; there might be a nickname that is shorter and easier to type, like **ed**. Typically the name of a package is also the name of the program that resides in that package.

There is a single namespace for packages. Instead of setting up a second-level package system to isolate names of packages from each other, we simply say that package name conflicts are to be resolved by using long explanatory names. There are sufficiently few packages in the world that a mechanism to allow two packages to have the same name does not seem necessary. Note that for the most frequent use of package names, qualified names of symbols, name clashes between packages can be alleviated using relative names.

The syntax conventions for package names are the same as for symbols. When the reader sees a package name (as part of a qualified symbol name), alphabetic characters in the package name are converted to uppercase unless preceded by the "/" escape character or unless the package name is surrounded by "|" characters. When a package name is printed by the printer, if it does not consist of all uppercase alphabetics and non-delimiter characters, the "/" and "|" escape characters are used.

Package name lookup is currently case-insensitive, but it might be changed in the future to be case-sensitive. In any case you should not make two packages whose names differ only in alphabetic case.

Internally names of packages are strings, but the functions that require a package-name argument from the user accept either a symbol or a string. If you supply a symbol, its print-name is used, which has already undergone case conversion by the usual rules. If you supply a string, you must be careful to capitalize the string in the same way that the package's name is capitalized.

Note that **|Foo|:|Bar|** refers to a symbol whose name is "**Bar**" in a package whose name is "**Foo**". By contrast, **|Foo:Bar|** refers to a 7-character symbol with a colon in its name, and is interned in the current package. Following the convention used in the documentation for symbols, we show package names as being in lowercase, even though the name string is really in uppercase.

23.4.1.1 Invisible Packages

In addition to normal packages, there can be *invisible* packages. An invisible package has a name, but it is not entered into the system's table that maps package names to packages. An invisible package cannot be referenced via a qualified name (unless you set up a relative name for it) and cannot be used in such contexts as the **:use** keyword to **defpackage** and **make-package** (unless you pass the package object itself, rather than its name). Invisible packages are useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

23.4.2 Relative Package Names

See the section "Introduction to Package Names", page 645. In addition to the absolute package names (and nicknames) described there, packages can have *relative* names. If **p** is a relative name for package B, relative to package A, then in contexts where relative names are allowed and A is the contextually relevant package the name **p** can be used instead of **b**. The relative name mapping *belongs to* package A and defines a new name (**p**) *for* package B. It is important not to confuse the package that the name is relative to with the package that is named.

Relative names are established with the **:relative-names** and **:relative-names-for-me** options to **defpackage** and **make-package**. For example, to be able to refer to symbols in the **common-lisp** package print with the prefix **lisp:** instead of **cl:** when they need a package prefix (for instance, when they are shadowed), you would use **:relative-names** like this:

```
(defpackage my-package (:use cl)
                       (:shadow error)
                       (:relative-names (lisp common-lisp)))

(let ((*package* (find-package 'my-package)))
  (print (list 'my-package::error 'cl:error)))
```

Then, when the current package (that is the value of ***package***) is **my-package**, you can refer to the **common-lisp** package as **lisp** both when reading and when printing. You can also use the **pkg-add-relative-name** function to establish a relative name. The **pkg-delete-relative-name** function removes a relative name.

There are two important differences between relative names and absolute names: relative names are recognized only in certain contexts, and relative names can "shadow" absolute names. One application for relative names is to replace one package by another. Thus if a program residing in package A normally refers to the **thermodynamics** package, but for testing purposes we would like it to use the **phlogiston** package instead, we can give A a relative name mapping from the name **thermodynamics** to the **phlogiston** package. This relative name shadows the absolute name **thermodynamics**.

Another application for relative names is to ease the establishment of a family of mutually dependent packages. For example, if you have three packages named **algebra, rings**, and **polynomials**, these packages might refer to each other so frequently that you would like to use the nicknames **a**, **r**, and **p** rather than spelling out the full names each time. It would obviously be bad to use up these one-letter names in the system-wide space of package names; what if someone else has a program with two packages named **reasoning** and **truth-maintenance**, and would like to use the nicknames **r** and **t**? The solution to this name conflict is to make the abbreviated names be relative names defined in the **algebra, rings**, and **polynomials** packages. These abbreviations are seen by references emanating from those packages, and there is no conflict with other abbreviations defined by other packages.

An extension of the shadowing application for relative names is to set up a complete family of packages parallel to the normal one, such as **experimental-global** and **experimental-user**. Within this family of packages you establish relative name mappings so that the usual names such as **global** and **user** can be used. Certain system utility programs work this way.

When package A uses package B, in addition to inheriting package B's external symbols, any relative name mappings established by package B are inherited. In the event of a name conflict between relative names defined directly by A and inherited relative names, the inherited name is ignored. The results are unpredictable if two relative name mappings inherited from two different packages conflict.

The Lisp system does not itself use relative names, so a freshly booted Lisp Machine contains no relative-name mappings.

Relative names are recognized in the following contexts:

- Qualified symbol names – The package name before the colon is relative to the package in which the symbol is being read (the value of the variable ***package***). The printer prefers a relative package name to an absolute package name when it prints a qualified symbol name.

- Package references in package-manipulating functions – For example, the package names in the **:use** option to **defpackage** and in the first argument to **use-package** can be relative names. All such relative names are relative to the value of the variable ***package***.

- Package arguments that default to the current package – The functions **intern**, **intern-local**, **intern-soft**, **intern-local-soft**, **unintern**, **export**, **unexport**, **import**, **shadow**, **shadowing-import**, **use-package**, and **unuse-package** all take an optional second argument that defaults (except in the case of **unintern**) to the current package. If supplied, this argument can be a package, an absolute name of a package, or a relative name of a package. All such relative names are relative to the value of the variable ***package***.

Relative names are not recognized in "global" contexts, where there is no obvious contextual package to be relative to, such as:

- File attribute lists ("-*-" lines)

- Package names requested from you as part of error recovery, or in commands such as the Set Package (m-X) editor command.

- The **pkg-find-package** function (unless its optional third argument is specified).

- Package arguments to the **zl:mapatoms**, **zl:pkg-goto**, **describe-package**, and **pkg-kill** functions.

- Package specifiers in the **do-symbols**, **do-local-symbols**, and **do-external-symbols** special forms, and the **interned-symbols** and **local-interned-symbols loop** iteration paths.

When a package object is printed, if it has a relative name (relative to the value of ***package***) that differs from its absolute name, both names are printed.

23.4.3 Qualified Package Names

23.4.3.1 Introduction to Qualified Package Names

Often it is desirable to refer to an external symbol in some package other than the current one. You do this through the use of a *qualified name*, consisting of a package name, then a colon, then the name of the symbol. This causes the symbol's name to be looked up in the specified package, rather than in the current one. For example, **editor:buffer** refers to the external symbol named **buffer** of the package named **editor**, regardless of whether there is a symbol named **buffer** in the current package. If there is no package named **editor**, or if no symbol named **buffer** is present in **editor** or if **buffer** is an internal symbol of **editor**, an error is signalled.

On rare occasions, you might need to refer to an *internal* symbol of some package other than the current one. It is invalid to do this with the colon qualifier, since accessing an internal symbol of some other package is usually a mistake. See the section "Specifying Internal and External Symbols in Packages", page 649. However, this operation is valid if you use "::" as the separator in place of the usual colon. If the reader sees **editor::buffer**, the effect is exactly the same as reading **buffer** with ***package*** temporarily rebound to the package whose name is **editor**. This special-purpose qualifier should be used with caution.

Qualified names are implemented in the Lisp reader by treating the colon character (:) specially. When the reader sees one or two colons preceded by the name of a package, it reads in the next Lisp object with ***package*** bound to that package. Note that the next Lisp object need not be a symbol; the printed representation of any Lisp object can follow a package prefix. If the object is a list, the effect is exactly as if every symbol in that list had been written as a qualified name, using the prefix that appears in front of the list. When a qualified name is among the elements of the list, the package name in the second package prefix is taken relative to the package selected by the first package prefix. The internal/external mode is controlled entirely by the innermost package prefix in effect.

23.4.3.2 Specifying Internal and External Symbols in Packages

To ease the transition for people whose programs are not yet organized according to the distinction between internal and external symbols, a package can be set up so that the ":" type of qualified name does the same thing as the "::" type. This is controlled by the package that appears before the colon, not by the package in which the whole expression is being read. To set this attribute of a package, use the **:colon-mode** keyword to **defpackage** and **make-package**. **:external** causes ":" to access only external symbols. See the section "Qualified Names of Symbols", page 650.. See the section "Qualified Package Names as Interfaces", page 649. **:internal** causes ":" to behave the same as "::", accessing all symbols. Note that **:internal** mode is compatible with **:external** mode except in cases where an error would be signalled. The default mode is **:internal** and all predefined system packages are created with this mode. In Common Lisp the default mode is **:external**.

23.4.3.3 Qualified Package Names as Interfaces

See the section "How the Package System Allows Symbol Sharing", page 636. In the example of the blocks world and the robot arm, a program in the **blocks** package could call a function named **go-up** defined in the **arm** package by calling **arm:go-up**. **go-up** would be listed among the external symbols of **arm**, using

:export in its **defpackage**, since it is part of the interface allowing the outside world to operate the arm. If the **blocks** program uses qualified names to refer to functions in the **arm** program, rather than sharing symbols as in the original example, then the possibility of name conflicts between the two programs is eliminated.

See the section "Example of the Need for Packages", page 636. Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function, it would simply call **arpa:allocate-pbuf**, assuming that function had been exported. If it was not exported (because **arpa** thought no one from the outside had any business calling it), the **chaos** program would call **arpa::allocate-pbuf**.

23.4.3.4 Qualified Names of Symbols

The printer uses qualified names when necessary. (The **princ** function, however, never prints qualified names for symbols.) The goal of the printer (for example, the **prin1** function) when printing a symbol is to produce a printed representation that the reader turns back into the same symbol. When a symbol that is accessible in the current package (the value of ***package***) is printed, a qualified name is not used, regardless of whether the symbol is present in the package. This happens for one of three reasons: because this is its home package, is present because it was imported, or is not present but was inherited. When an inaccessible symbol is printed, a qualified name is used. The printer chooses whether to use ":" or "::" based on whether the symbol is internal or external and the **:colon-mode** of its home package. The qualified name used by the printer can be read back in and yields the same symbol. If the inaccessible symbol were printed without qualification, the reader would translate that printed representation into a different symbol, probably an internal symbol of the current package.

The qualified name used by the printer is based on the symbol's home package, not on the path by which it was originally read (which of course cannot be known). Suppose **foo** is an internal symbol of package A, has been imported into package B, and has then been exported from package B. If it is printed while ***package*** is neither A nor B, nor a package that uses B, the name printed is **a::foo**, not **b:foo**, because **foo**'s home package is A. This is an unlikely case, of course.

In addition to the simplest printed representation of a symbol, its name standing by itself, there are four forms of qualified name for a symbol. These are accepted by the reader and are printed by the printer when necessary; except when printing an uninterned symbol, the printer prints some printed representation that yields the same symbol when read. The following table shows the four forms of qualified name, assuming that the **foo** package specifies **:colon-mode :external**. If **foo** specifies **:colon-mode :internal**, as is currently the default, the first and second forms are equivalent.

foo:bar	When read, looks up **bar** among the external symbols of the package named **foo**. Printed when the symbol **bar** is external in its home package **foo** and is not accessible in the current package.
foo::bar	When read, interprets **bar** as if **foo** were the current package. Printed when the symbol **bar** is internal in its home package **foo** and is not accessible in the current package.
:bar	When read, interprets **bar** as an external symbol in the **keyword** package. Printed when the home package of the symbol **bar** is **keyword**.
#:bar	When read, creates a new uninterned symbol named **bar**. Printed when the symbol named **bar** has no home package.

23.4.3.5 Multilevel Qualified Package Names

Due to shadowing by relative names, a given package might sometimes be inaccessible. In this case a multilevel qualified name, containing more than one package prefix, can be used.

Suppose packages **moe**, **larry**, **curly**, and **shemp** exist. For its own reasons, the **moe** package uses **curly** as a relative name for the **shemp** package. Thus, when the current package is **larry** the printed representation **curly:hair** designates a symbol in the **curly** package, but when the current package is **moe** the same printed representation designates a symbol in the **shemp** package.

If the **moe** package is current and the symbol **hair** in the **curly** package needs to be read or printed, the printed representation **curly:hair** cannot be used since it refers to a different symbol. If **curly** had a nickname that is not also shadowed by a relative name it would be used, but suppose there is no nickname. In this case the only possible way to refer to that symbol is with a multilevel qualified name. **larry:curly:hair** would work, since the **larry:** escapes from the scope of **moe**'s relative name. The printer actually prefers to print **global:curly:hair** because of the way it searches for a usable qualified name.

23.5 Examples of Symbol Sharing Among Packages

See the section "How the Package System Allows Symbol Sharing", page 636. Consider again the example of the robot arm in the blocks world. Two separate programs, written by different people, interact with each other in a single Lisp environment. The arm-control program resides in a package named **arm**, and the blocks-world program resides in a package named **blocks**. The operation of the two programs requires them to interact. For example, to move a block from one place to another the **blocks** program calls functions in the **arm** program with names like **raise-arm, move-arm,** and **grasp**. To find the edges of the table, the **arm** program accesses variables of the **blocks** program.

Communication between the two programs requires that they both know about certain objects. Usually these objects are the sort that have names (for example, functions or variables). The names are symbols. Thus each program must be able to name some symbols and to know that the other program is naming the same symbols.

Let us consider the case of the function **grasp** in the arm-control program, which the blocks-world program must call in order to pick up a block with the arm. The **grasp** function is named by the symbol **grasp** in the **arm** package. Assume that we are not going to use either of the mechanisms (keywords and the **global** package) that make symbols available to *all* packages; we only want **grasp** to be shared between the two specific packages that need it. There are basically three ways provided by the package system for a symbol to be known by two separate programs in two separate packages.

1. If the **blocks** package *imports* the symbol **grasp** from the **arm** package, then both packages map the name **grasp** into the same symbol. The **blocks** package could be defined by:

   ```
   (defpackage blocks
           (:import-from arm grasp))
   ```

2. The **arm** package can *export* the symbol **grasp**, along with whatever other symbols constitute its interface to the outside world. If the **blocks** package *uses* the **arm** package, then both packages again map the name **grasp** into the same symbol. The package definitions would look like:

   ```
   (defpackage arm
           (:export grasp move-arm raise-arm ...))

   (defpackage blocks
           (:use arm global))
   ```

   Note that the **blocks** package must explicitly mention that it is using the **global** package as well as the **arm** package, since it is not letting its **:use** clause default. The difference between this method (the export method) and the first method (the import method) is that the list of symbols that is to constitute the interface is associated with the **arm** package, that is, the package that *provides* the interface, not the package that *uses* the interface.

3. In the third method, we do not have the two packages map the same name into the same symbol. Instead we use a different, longer name for the symbol in the blocks program than the name used by the arm program. This makes it clear, when reading the text of the blocks program, which

symbol references are connected with the interface between the two programs. These longer names are called *qualified names.* Again, the **arm** package defines the interface:

```
(defpackage arm
        (:export grasp move-arm raise-arm ...))
```

A fragment of the blocks-world program might look like

```
(defun pick-up (block)
  (clear-top block)
  (arm:grasp (block-coordinates block :top))
  (arm:raise-arm))
```

arm:grasp and **arm:raise-arm** are qualified names. **pick-up**, **block**, **clear-top**, and **block-coordinates** are internal symbols of the blocks-world program. **defun** is inherited from the **global** package. **:top** is a keyword. Note that although the two programs do not use the same names to refer to the same symbol, the names they use are related in an obvious way, avoiding confusion. The package system makes no provision for the same symbol to be named by two completely arbitrary names.

23.6 Consistency Rules for Packages

Package-related bugs can be very subtle and confusing: the program is not using the same symbols as you think it is using. The package system is designed with a number of safety features to prevent most of the common bugs that would otherwise occur in normal use. This might seem overprotective, but experience with earlier package systems has shown that such safety features are needed.

In dealing with the package system, it is useful to keep in mind the following consistency rules, which remain in force as long as the value of ***package*** is not changed by you or your code:

- *Read-Read consistency:* Reading the same print name always gets you the same (**eq**) symbol.

- *Print-Read consistency:* An interned symbol always prints as a sequence of characters that, when read back in, yields the same (**eq**) symbol.

- *Print-Print consistency:* If two interned symbols are not **eq**, then their printed representations cannot be the same sequence of characters.

These consistency rules remain true in spite of any amount of implicit interning caused by typing in Lisp forms, loading files, and so on. This has the important implication that results are reproducible regardless of the order of either loading files or typing in symbols. The rules can only be violated by explicit action: changing the value of ***package***, forcing some action by continuing from an error, or calling a function that makes explicit modifications to the package structure (**unintern**, for example).

To ensure that the consistency rules are obeyed, the system ensures that certain aspects of the package structure are chosen by conscious decision of the programmer, not by accidents such as which symbols happen to be typed in by a user. External symbols, the symbols that are shared between packages without being explicitly listed by the "accepting" package, must be explicitly listed by the "providing" package. No reference to a package can be made before it has been explicitly defined.

23.7 Package Name-Conflict Errors

23.7.1 Introduction to Package Name-Conflict Errors

A fundamental invariant of the package system is that within one package any particular name can refer to only one symbol. A *name conflict* is said to occur when more than one candidate symbol exists and it is not obvious which one to choose. If the system does not always choose the same way, the read-read consistency rule would be violated. For example, some programs or data might have been read in under a certain mapping of the name to a symbol. If the mapping changes to a different symbol, then additional programs or data are read, the two programs do not access the same symbol even though they use the same name. Even if the system did always choose the same way, a name conflict is likely to result in a different mapping from names to symbols than you expected, causing programs to execute incorrectly. Therefore, any time a name conflict occurs, an error is signalled. You can continue from the error and tell the package system how to resolve the conflict.

Note that if the same symbol is accessible to a package through more than one path, for instance as an external of more than one package, or both through inheritance and through direct presence in the package, there is no name conflict. Name conflicts only occur between distinct symbols with the same name.

See the section "Shadowing Symbols", page 642. As discussed there, the creator of a package can tell the system in advance how to resolve a name conflict through the use of *shadowing*. Every package has a list of shadowing symbols. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. A name conflict involving a shadowing symbol is always resolved in favor of the shadowing symbol, without signalling an error (except for one exception involving **import**). The **:shadow** and **:shadowing-import** options to **defpackage** and **make-package** can be used to declare shadowing symbols. The functions **shadow** and **shadowing-import** can also be used.

23.7.2 Checking for Package Name-Conflict Errors

Name conflicts are detected when they become possible, that is, when the package structure is altered. There is no need to check for name conflicts during every name lookup. The functions **use-package, import**, and **export** check for name conflicts.

Using a package makes the external symbols of the package being used accessible to the using package; each of these symbols is checked for name conflicts with the symbols already accessible.

Importing a symbol adds it to the internals of a package, checking for a name conflict with an existing symbol either present in the package or accessible to it. **import** signals an error even if there is a name conflict with a shadowing symbol, because two explicit directives from you are inconsistent.

Exporting a symbol makes it accessible to all the packages that use the package from which the symbol is exported. All of these packages are checked for name conflicts. **(export** *s p***)** does **(intern-soft** *s q***)** for each package *q* in **(package-used-by-list** *p***)**. Note that in the usual case of exporting symbols only during the initial definition of a package, there are no users of the package yet and the name-conflict checking takes no time.

intern does not need to do any name-conflict checking, because it never creates a new symbol if there is already an accessible symbol with the name given.

Note that the function **intern-local** can create a new symbol with the same name as an already accessible symbol. Nevertheless, **intern-local** does not check for name conflicts. This function is considered to be a low-level primitive and indiscriminate use of it can cause undetected name conflicts. Use **import, shadow**, or **shadowing-import** for normal purposes.

shadow and **shadowing-import** never signal a name-conflict error, because by calling these functions the user has specified how any possible conflict is to be resolved. **shadow** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the specified name is accessible, and if so whether it is directly present in the package or inherited; in the latter case a new symbol is created to shadow it. **shadowing-import** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the same name is accessible; if so it is shadowed by the new symbol, which implies that it must be uninterned (with **unintern**) if it was directly present in the package.

unuse-package, unexport, and **unintern** (when the symbol being removed is not a shadowing symbol) do not need to do any name-conflict checking, because they only remove symbols from a package; they do not make any new symbols accessible.

unintern of a shadowing symbol can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol **x**, and B and C each contain external symbols named **x**, then uninterning **x** from A reveals a name conflict between **b:x** and **c:x** if those two symbols are distinct. In this case **unintern** signals an error.

23.7.3 Resolving Package Name-Conflict Errors

Aborting from a name-conflict error leaves the original symbol accessible. Package functions always signal name-conflict errors before making any change to the package structure. Note: when multiple changes are to be made, for example when exporting a list of symbols, it is valid for each change to be processed separately, so that aborting from a name conflict caused by the second symbol in the list does not unexport the first symbol in the list. However, aborting from a name-conflict error caused by exporting a single symbol does not leave that symbol accessible to some packages and inaccessible to others; exporting appears as an atomic operation.

Continuing from a name-conflict error offers you a chance to resolve the name conflict in favor of either of the candidates. This can involve shadowing or uninterning. Another possibility that is offered to you is to merge together the conflicting symbols' values, function definitions, and property lists in the same way as **globalize**. This is useful when the conflicting symbols are not being used as objects, but only as names for functions (or variables, or flavors, for example). You are also offered the choice of simply skipping the particular package operation that would have caused a name conflict.

A name conflict in **use-package** between a symbol directly present in the using package and an external symbol of the used package can be resolved in favor of the first symbol by making it a shadowing symbol, or in favor of the second symbol by uninterning the first symbol from the using package. The latter resolution is dangerous if the symbol to be removed is an external symbol of the using package, since it ceases to be an external symbol.

A name conflict in **use-package** between two external symbols inherited by the using package from other packages can be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol.

A name conflict in **export** between the symbol being exported and a symbol already present in a package that would inherit the newly exported symbol can be resolved in favor of the exported symbol by uninterning the other one, or in favor of the already present symbol by making it a shadowing symbol.

A name conflict in **export** or **unintern** due to a package inheriting two distinct symbols with the same name from two other packages can be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol, just as with **use-package**.

A name conflict in **import** between the symbol being imported and a symbol inherited from some other package can be resolved in favor of the symbol being imported by making it a shadowing symbol, or in favor of the symbol already accessible by not doing the **import**. A name conflict in **import** with a symbol already present in the package can be resolved by uninterning that symbol, or by not doing the **import**.

Good user-interface style dictates that **use-package** and **export**, which can cause many name conflicts simultaneously, first check for all of the name conflicts before presenting any of them to you. You can then choose to resolve all of them wholesale, or to resolve each of them individually, requiring considerable interaction but permitting different conflicts to be resolved different ways.

23.8 External-only Packages and Locking

A package can be *locked*, which means that any attempt to add a new symbol to it signals an error. Continuing from the error adds the symbol.

When reading from an interactive stream, such as a window, the error for adding a new symbol to a locked package does not go into the Debugger. Instead it asks you to correct your input, using the input editor. You cannot add a new symbol to a locked package just by typing its name; you must explicitly call **intern**, **export**, or **globalize**.

A package can be declared *external-only*. This causes any symbol added to the package to be exported automatically. Since exporting of symbols should be a conscious decision, when you create an external-only package it is automatically locked. Any attempt to add a new symbol to an external-only package signals an error because it is locked. If adding the symbol would cause a name conflict in some package that uses the package to which the symbol is being added, the error message mentions that fact. Continuing from the error adds the symbol anyway. In the event of name conflicts, appropriate proceed types for resolving name conflicts are offered.

To set up an external-only package, it can be temporarily unlocked and then the desired set of symbols can be interned in it. Unlocking an external-only package disables name-conflict checking, since the system (perhaps erroneously) assumes you know what you are doing. The **global** package is external-only and locked. Its contents are initialized when the system is built by reading files containing the desired symbols with ***package*** bound to the **global** package object, which is temporarily unlocked. The **system** package is external-only, locked, and initialized the same way.

23.9 Package Functions, Special Forms, and Variables

Packages are represented as Lisp objects. A package is a structure that contains various fields and a hash table that maps from names to symbols. Most of the structure field accessor functions for packages are only used internally by the package system and are not documented.

The **packagep** function returns **t** if its argument is a package object. This is equivalent to **(typep** *obj* **'package)**.

Many of the functions that operate on packages accept either an actual package or the name of a package. A package name can be either a string or a symbol.

Common Lisp naming convention uses a prefix of "**package-**" on names that do not already contain the word **package**. Many of the Zetalisp functions and variables associated with packages have names that begin with "**pkg-**". This naming convention is considered obsolete.

23.9.1 The Current Package

The current package is the value of the variable ***package***.

The following two functions change the current package:

zl:pkg-goto *pkg*	Make *pkg* the current package.
zl:pkg-bind *pkg body*	Evaluate *body* with *pkg* the current package.

23.9.2 Defining a Package

The **defpackage** special form is the preferred way to create a package. A **defpackage** form is treated as a *definition* form by the editor; hence the Edit Definition (m-.) command can find package definitions.

Typically you put a **defpackage** form in its own file, separate from the rest of a program's source code. The reason to use a separate file is that a package must be defined before it can be used. In order to compile, load, or edit your program, the package in which its symbols are to be read must already be defined. Typically the file containing the **defpackage** is read in the **user** package, while all the rest of the files of your program are read in your own private package.

When a large program consisting of multiple source files is maintained with the system system, one source file typically contains nothing but a **defpackage** form and a **defsystem** form. (Occasionally a few other housekeeping forms are present.) This file is called the *system declaration file.* The packages and systems built into the initial Lisp system are defined in two files: sys:sys;pkgdcl defines all the packages while sys:sys;sysdcl defines all the systems. See the section "Maintaining Large Programs" in *Program Development Utilities.*

In the simplest cases, where no nontrivial **defpackage** options are required, the **defpackage** form can be omitted and no separate file is required. All the information required to create your package is contained in the file attribute list of the file containing your program. See the section "Specifying Packages in Programs", page 643.

The **make-package** function is available as the primitive way to create package objects.

23.9.2.1 Functions for Defining a Package

defpackage *name options*	Define a package named *name*.
make-package *name*	Primitive subroutine called by **defpackage**. **defpackage** should be used in new programs.
pkg-kill *package*	Kill *package*.

23.9.3 Mapping Names to Symbols

The name of a symbol is a string, corresponding to the printed representation of that symbol with quoting characters removed. Mapping the name of a symbol into the symbol itself is called *interning*, for historical reasons. Interning is only meaningful with respect to a particular package, since packages are name-to-symbol mappings. Unless a package is explicitly specified, the current package is assumed.

There are four functions for interning: **intern, intern-soft, intern-local**, and **intern-local-soft**. Each function takes two arguments and returns two values. The arguments are a name and a package. The name can be a string or a symbol. The package argument can be a package, the name of a package as a string or a symbol, or **nil** or unsupplied, in which case the current package (the value of ***package***) is used by default.

The **-soft** functions do not create new symbols, but only find existing symbols. The other two functions add a new symbol to the package if no existing symbol with the specified name is found. When adding a new symbol, if the name argument is a string, a new symbol is created and its home package is made to be the specified package. If the name argument is a symbol, that symbol is used as the new symbol. If it has a home package, it is not changed, but if it does not have a home package its home package is set to the package to which it was just added.

The **-local** functions only look for symbols present in the package; they do not search through inherited symbols. The other two functions see all accessible symbols.

The first value is the symbol that was found or created, or **nil** if no symbol was found and a **-soft** function was called. The second value is a flag that takes on one of the following values:

nil	No preexisting symbol was found. If the function called was not a **-soft** version, a new internal symbol was added to the package.
:internal	An existing internal symbol was found to be present in the package.
:external	An existing external symbol was found to be present in the package.
:inherited	An existing symbol was found to be inherited by the package. This symbol is necessarily external in the package from which it was inherited, and cannot be external in the package being searched.

Note that the first value should not be used as a flag to detect whether or not a symbol was found, since the *false* value, **nil**, is a symbol. The second value must be used for this purpose. The **-soft** functions return both values **nil** if they do not find a symbol.

Note: interning is sensitive to case; that is, it considers two character strings different even if the only difference is one of uppercase versus lowercase (unlike most string comparisons elsewhere in Genera). Symbols are converted to uppercase when you type them in because the reader converts the case of characters in the printed representation of symbols; the characters are converted to uppercase before **intern** is ever called. So if you call **intern** with a lowercase **"foo"** and then with an uppercase **"foo"**, you do not get the same symbol.

23.9.3.1 Functions That Map Names to Symbols

intern *string*	Finds or creates a symbol named *string*.
zl:intern *string*	Finds or creates a symbol named *string*.
intern-local *string*	Finds or creates *string* in the current package.
intern-soft *string*	Finds a symbol *string* that is accessible to the current package.
intern-local-soft *string*	Finds a symbol *string* in the current package.
find-all-symbols *string*	Finds a symbol *string* by searching all packages.
unintern *symbol*	Removes *symbol.*
zl:remob *symbol*	Removes *symbol.*

23.9.4 Functions That Find the Home Package of a Symbol

symbol-package *symbol*	Return the package in which *symbol* resides
sys:package-cell-location *symbol*	Return a locative pointer to *symbol*'s package cell.
keywordp *object*	Check if *object* is a symbol in the keyword package.

23.9.5 Mapping Between Names and Packages

package-name *package*	Returns the name of *package* as a string.
zl:pkg-name *package*	Gets the primary name of a package as a string.
find-package *package*	Returns the package object whose name is *package*.
pkg-find-package *x*	Tries to interpret *x* as a package.
rename-package *pkg new-name*	Replaces the old name of *pkg* with *new-name*.

23.9.6 Package Iteration

The following functions allow you to operate on the symbols in a package.

zl:mapatoms *function*	Applies *function* to all symbols in current package.
zl:mapatoms-all *functions*	Applies *function* to all symbols in all packages.
do-symbols *variable body*	Evaluates *body* with *variable* bound to each symbol accessible in current package in turn.
do-local-symbols *variable body*	Evaluates *body* with *variable* bound to those symbols present in the current package in turn.
do-external-symbols *variable body*	Evaluates *body* with *variable* bound to those symbols exported from the current package in turn.
do-all-symbols *variable body*	Evaluate *body* with *variable* bound to each symbol in all packages in turn.

See the section "Iteration Paths", page 553. This section contains a discussion of the **cl:interned-symbols** and **cl:local-interned-symbols loop** iteration paths.

23.9.7 Interpackage Relations

pkg-add-relative-name *from-package name to-package*	Allows *from-package* to use *name* to refer to *to-package*.
pkg-delete-relative-name *from-package name*	Removes *name*.
package-use-list *package*	Lists other packages used by *package*.
package-used-by-list *package*	Lists the packages that use *package*.
use-package *packages-to-use*	Lets *packages-to-use* be used by the current package.

unuse-package *packages-to-unuse*	Remove *packages-to-unuse* from the use-list.

23.9.8 Functions That Import, Export, and Shadow Symbols

export *symbols*	Makes *symbols* external.
unexport *symbols*	Makes *symbols* internal.
import *symbols*	Makes *symbols* internal.
shadowing-import *symbols*	Makes *symbols* internal.
shadow *symbols*	Makes *symbols* internal.
package-external-symbols *package*	Lists external symbols.
package-shadowing-symbols *package*	Lists the shadowing symbols in *package*.

23.9.9 Package "Commands"

describe-package *package*	Displays the attributes of *package*.
where-is *pname*	Displays a description of each symbol named *pname* with its home package.
globalize *name*	Exports *name*.

There is a subtle pitfall in the interaction between **globalize** and the binary files output by the compiler. Because of this it is best to use a string, rather than a symbol, as the argument to **globalize** in files that are to be compiled. Suppose a file contains the following form at top level:

```
(eval-when (compile load eval)
  (globalize 'si:rumpelstiltskin))
```

If the file is loaded without being compiled, the form is read and evaluated in the obvious fashion. **rumpelstiltskin** is read as the symbol by that name in the **si** package, that symbol is passed to the **globalize** function, and the symbol is moved to the **global** package. Now suppose the file is compiled. Again **rumpelstiltskin** is read as the symbol by that name in the **si** package. The **eval-when** causes the compiler first to evaluate the **globalize** form, and then to place a representation of the form into its output file. But at the time the output file is being generated, the symbol **rumpelstiltskin** is global; the compiler no longer has any way to know that it came from the **si** package. When the binary file is loaded, it globalizes the symbol **rumpelstiltskin** in the current package, not the one in the **si** package as the programmer intended. Furthermore, if at compile time there was a **rumpelstiltskin** symbol in the current package, the compile-time **globalize** turns that symbol into a shadowing symbol. When the binary file is loaded, it tries to refer to the symbol **rumpelstiltskin** in the **global** package, which gets an error since the **global** package is locked. The same pitfall can arise without the use of **eval-when** if the file being compiled was previously loaded into the Lisp that compiled it, perhaps for test purposes.

23.9.10 System Packages

The following are some of the packages initially present in the Lisp world. New packages will be added to this list from time to time. The list is presented in "logical" order, with the most important or interesting packages first. A number of packages that are not of general interest have been omitted from the list for the sake of brevity.

cl, **common-lisp** or **common-lisp-global**
: Contains the global symbols of Common Lisp. Inside of Common Lisp this package is called **lisp**. **common-lisp-global** does not inherit symbols from any other package.

zl or **global**
: Contains the global symbols of the Zetalisp language, including function names, variable names, special form names, and so on. All symbols in **global** are supposed to be documented. **global** does not inherit symbols from any other package.

scl or **symbolics-common-lisp**
: Contains the symbols that make up Symbolics extensions to Common Lisp. **symbolics-common-lisp** uses **common-lisp-global.**

keyword
: Contains keyword symbols. **keyword** has a blank nickname so that keywords print as **:foo** rather than **keyword:foo**. **keyword** does not inherit symbols from any other package.

cl-user or **common-lisp-user**
: The default package for user programs that do not have their own package. When first booted the Symbolics Lisp Machine uses the **cl-user** package to read expressions typed in by the user. Inside of Common Lisp, **cl-user** is called **user**. **cl-user** uses **common-lisp-global.**

zl-user or **zetalisp-user**
: The default package for Zetalisp user programs. (See the section "Set Lisp Context Command" in *User's Guide to Symbolics Computers.*) Inside Zetalisp, **zl-user** is called **user**. **zl-user** uses **global.**

sys or **system**
: Contains symbols shared among various system programs. **system** is for symbols global to the Genera "operating system", while **common-lisp-global** and **global** are for symbols global to the Symbolics Common Lisp language.

si or **system-internals**
: Most of the programs that implement the Lisp language and

	operating system are in the **system-internals** package. **system-internals** is one of the packages that uses **system**. The externally advertised symbols of these programs are in **system** or **global**.
compiler	Contains the Lisp compiler. **compiler** is one of the packages that use **system**.
dbg or **debugger**	Contains the condition system and the debugger. **debugger** is one of the packages that use **system**.
cp or **command-processor**	Contains the Command Processor.
dw or **dynamic-windows**	Contains the dynamic window system. **dw** uses **system**.
tv	Contains the window system window system substrate. **tv** is one of the packages that use **system**.
zwei	Contains the editor and Zmail.
mailer	Contains the Symbolics Store-and-Forward mailer.
fs or **file-system**	Contains pathnames and the generic file access system. **file-system** is one of the packages that use **system**.
lmfs	Contains the Symbolics Lisp Machine file storage system. **lmfs** is one of the packages that use **system**.
format	Contains the function **format** and its associated subfunctions.
hardcopy	Contains the functions for sending output to printers.
net or **network**	Contains the external interfaces to the generic network system. **network** is one of the packages that uses **system**. Each network implementation and network-related program has its own package, which uses **network**.
neti or **network-internals**	Contains the programs that implement the generic network system. **network-internals** uses **network** and **system**.
chaos	Contains the Chaosnet control program. **chaos** is one of the packages that use **network** and **system**.
fonts	Contains the names of all fonts. **fonts** does not inherit symbols from any other package.

The following variables have the most important packages as their values.

zl:pkg-global-package	The **global** package.

sys:pkg-keyword-package	The **keyword** package.
zl:pkg-system-package	The **system** package.

23.10 Package-related Conditions

These are the most basic package-related conditions. There are other conditions built on these, but most programmers should not need to deal with them.

sys:package-error	Basic error condition for packages.
sys:package-not-found	Package lookup did not find a package with the specified name.
sys:external-symbol-not-found	The specified symbol is not external.
sys:package-locked	There was an attempt to intern a symbol in a locked package.
sys:name-conflict	Any sort of name conflict.

23.11 Multipackage Programs

Usually, each independent program occupies one package. But large programs, such as MACSYMA, are usually made up of a number of subprograms, and each subprogram might be maintained by a different person or group of people. We would like each subprogram to have its own namespace, since the program as a whole has too many names for anyone to remember. The package system can provide the same benefits to programs that are part of the same superprogram as it does to programs that are completely independent.

Putting each subprogram into its own package is easy enough, but it is likely that a fair number of functions and symbols should be shared by all of MACSYMA's subprograms. These would be internal interfaces between the different subprograms.

A package named **macsyma** can be defined and each of the internal interface symbols can be exported from it. Each subprogram of MACSYMA has its own package, which uses the **macsyma** package in addition to any other packages it uses. Thus the interface symbols are accessible to all subprograms, through package inheritance. These interface symbols typically get their function definitions, variable values, and other properties from various subprograms read into the various internal MACSYMA packages, although there is nothing wrong with also putting a subprogram directly into the **macsyma** package. This is similar to the way the Lisp system works; the **global** package exports a large number of symbols, which get their values, definitions, and so on from programs residing in other packages that use **global**, such as **system-internals** or **compiler**.

It is also often convenient for the **macsyma** package to supply relative names that can be used by the various subprograms to refer to each other's packages. This allows package name abbreviations to be used internally to MACSYMA without contaminating the external environment.

The system declaration file for MACSYMA would then look something like the following:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified names

;The integration package based on the Risch algorithm
(defpackage risch
            (:use macsyma global))

;The integration package based on pattern matching
(defpackage sin
            (:use macsyma global))

;Interface to the operating system.  This uses the SYSTEM package
;because it needs many system-dependent functions and constants.
;This package also has a local nickname because its primary name
;is so long.
(defpackage macsyma-system-interface
            (:relative-names-for-me (macsyma sysi))
            (:use macsyma system global))
```

You can break the interface symbols down into separate categories. For instance, you might want to separate internal symbols used only inside MACSYMA from symbols that are also useful to the outside world. The latter symbols clearly should be externals of the **macsyma** package. You could create an additional package named **macsyma-internals** that exports all the symbols that are interfaces between different subprograms of MACSYMA but are not for use by the outside world. In this case we would have:

```
(defpackage risch
            (:use macsyma-internals macsyma global))
```

A program in the outside world that needed to use parts of MACSYMA would either use qualified names such as **macsyma:solve** or would include **macsyma** in the **:use** option in its package definition.

The interface symbols can be broken down into even more categories. Each sub-package can have its own list of exported symbols, and can use whichever other subpackages it depends on. The subset of these exported symbols that are also useful to the outside world can be exported from the **macsyma** package as well. In this case our example system declaration file would look something like:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export solve integrate ...)
            (:colon-mode :external))     ;error-checking in qualified names

;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))

;The macsyma interpreter
(defpackage meval
            (:export meval mprint ...))
```

The symbol **integrate** exported by the **macsyma** package and the symbol **integrate** exported by the **risch** package are the same symbol, because **risch** inherits it from **macsyma**.

Sometimes you can get involved in forward references when setting up this sort of package structure. In the above example, **risch** needs to use **rat**, hence **rat** was defined first. If **rat** also needed to use **risch**, there would be no way to write the package definitions using only **defpackage**. In this case you can explicitly call **use-package** after both packages have been defined. For example:

```
;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))      ;also uses risch

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))
```

```
;Now complete the forward references
(use-package 'risch 'rat)
```

An analogous issue arises when using **:import**.

Now, the **risch** program and the **sin** program both do integration, and so it would be natural for each to have a function called **integrate**. From inside **sin**, **sin**'s **integrate** would be referred to as **integrate** (no prefix needed), while **risch**'s would be referred to as **risch::integrate** or as **risch:integrate** if **risch** exported it (which is likely). Similarly, from inside **risch**, **risch**'s own **integrate** would be called **integrate**, whereas **sin**'s would be referred to as **sin::integrate** or **sin:integrate**.

If **sin**'s **integrate** were a recursive function, you would refer to it from within **sin** itself, and would not have to type **sin:integrate** every time; you would just say **integrate**.

If the names **sin** and **risch** are considered to be too short to use up in the general space of package names, they can be made local abbreviations within MACSYMA's family of package through local names. The package definitions would be

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
           (:export meval mprint ptimes ...)
           (:colon-mode :external))    ;error-checking in qualified names

;The integration package based on the Risch algorithm
(defpackage macsyma-risch-integration
           (:relative-names-for-me (macsyma risch))
           (:use macsyma global))

;The integration package based on pattern matching
(defpackage macsyma-pattern-integration
           (:relative-names-for-me (macsyma sin))
           (:use macsyma global))
```

From inside the **macsyma** package or any package that uses it the two integration functions would be referred to as **sin:integrate** and as **risch:integrate**. From anywhere else in the hierarchy, they could be called **macsyma:sin:integrate** and **macsyma:risch:integrate**, or **macsyma-pattern-integration:integrate** and **macsyma-risch-integration:integrate**.

PART VI.

Understanding Compatibility Issues

24. Introduction to Symbolics Common Lisp

Genera 7.0 provides three dialects of Lisp for you to use:

Symbolics Common Lisp	This is based on Common Lisp; it includes Common Lisp as well as all the advanced features of Zetalisp. Symbolics Common Lisp is the default dialect in Genera 7.0.
Zetalisp	The dialect of Lisp provided in all previous Symbolics releases.
Common Lisp	An implementation of Common Lisp as described in *Common Lisp the Language (CLtL)* by Guy Steele.

All three of these dialects have some underlying features in common:

- Both the interpreter and the compiler use lexical scoping in all dialects.
- Characters are represented as character objects in all three dialects.
- Row-major arrays are used by all three dialects.

Both Symbolics Common Lisp and Zetalisp use the interpreter, compiler, the same data structures, and other tools.

Syntactic differences between Common Lisp and Zetalisp are handled by Zetalisp reader/printer control variables, such as **ibase**, **base**, **readtable**, and **package**. In Common Lisp programs these variables appear under the names ***read-base***, ***print-base***, ***readtable***, and ***package***. The binding of these variables is controlled automatically by the system.

Most Zetalisp functions, special forms, and facilities are available in SCL. Some of them, such as the **defstruct** macro, have been modified to make them compatible with Common Lisp.

For a description of the differences between SCL and Common Lisp as described in the Digital Press edition of the *Common Lisp* manual (*CLM*) by Guy Steele: See the section "Compatibility with Common Lisp", page 675.

24.1 SCL Packages

SCL provides a separate set of packages for Common Lisp. When the two dialects have a feature in common, some of the symbols in these packages are identical to symbols in Zetalisp. Other symbols are specific to Common Lisp.

The **common-lisp** package contains all the symbols defined in Common Lisp, while the **symbolics-common-lisp** package contains those symbols plus the symbols that are Symbolics extensions to Common Lisp. The symbols in SCL can be found in both the **common-lisp** and **symbolics-common-lisp** packages.

The following packages are provided by SCL:

common-lisp	This package exports all symbols defined by Common Lisp, other than keywords. It is also known by the names **common-lisp-global**, **lisp**, and **cl**. All Common Lisp packages inherit from the **common-lisp** package. The Common Lisp name for this package is **lisp**.
symbolics-common-lisp	This package exports all the symbols that are either in Common Lisp or are Symbolics extensions to Common Lisp. Most of the internals used by SCL are in this package. It is also known by the name **scl**.
common-lisp-user	This is the default package for user programs. It is also known by the names **user** and **cl-user**. **common-lisp-user** inherits from **symbolics-common-lisp**. User programs should be placed in the **common-lisp-user** package, rather than the **common-lisp** package, to insulate them from the internal symbols of SCL. The Common Lisp name for this package is **user**.
common-lisp-system	This package exports a variety of 3600-specific architectural and implementational symbols. It is also known by the name **cl-sys**. In Zetalisp, some of these symbols are in **global** and some are in **system**. **common-lisp-user** does not inherit from **common-lisp-system**. The Common Lisp names for this package are **system** and **sys**.
gprint	This package contains portions of SCL concerned with the printing of Lisp expressions. It is not a standard Common Lisp package.
language-tools	This package contains portions of SCL concerned with Lisp code analysis and construction. It has the nickname **lt**. It is not a standard Common Lisp package.
zl	The name **zl** can be used in a Common Lisp program to refer to Zetalisp's **global** package. The name **zetalisp** is synonymous with **zl**.

zl-user	The name **zl-user** can be used in a Common Lisp program to refer to Zetalisp's **user** package.

SCL and Zetalisp share the same keyword package.

Common Lisp packages can be referred to by their Common Lisp names from Common Lisp programs, but not from Zetalisp programs. These names are relative names defined by the **common-lisp** package.

All Zetalisp packages can be referred to from a Common Lisp program. Those packages that have the same name as a Common Lisp package, such as **system** and **user**, can be referenced with a multilevel package prefix, for example, **zl:user:foo**. **zl-user:foo** is synonymous with **zl:user:foo**.

Packages can be used to shadow Common Lisp global symbols. For example, if you have a program in which you would like to use **merge** as the name of a function, you put the program in its own package (separate from **cl-user**), specify **:shadow merge** in the **defpackage**, and use **lisp:merge** to refer to the SCL **merge** function.

24.2 SCL and Symbolics Common Lisp Extensions

Most of the language features of Zetalisp that are not in Common Lisp are provided by SCL in the **symbolics-common-lisp** package. This includes such things as processes, **loop**, and flavors. In some cases (**string-append**, for example) these Zetalisp features have been modified to make them implementationally or philosophically compatible with Common Lisp. In most cases you can refer to the documentation for information about these features.

24.3 SCL and Common Lisp Files

The file attribute line of a Common Lisp file should be used to tell the editor, the compiler, and other programs that the file contains a Common Lisp program. The following file attributes are relevant:

Syntax	The value of this attribute can be Common-Lisp or Zetalisp. It controls the binding of the Zetalisp variable **readtable**, which is known as ***readtable*** in Common Lisp. The default syntax is Common-Lisp.
Package	**user** is the package most commonly used for Common Lisp programs. You can also create your own package. Note that the Package file attribute accepts relative package names, which means that you can specify **user** rather than **cl-user**.

The following example shows the attributes that should be in an SCL file's attribute line:

```
;;; -*- Mode:Lisp; Syntax:Common-Lisp; Package:USER -*-
```

25. Compatibility with Common Lisp

Some differences exist between the Symbolics implementation of Common Lisp and the language specification presented in Steele's *Common Lisp the Language* manual *(CLtL)*. This section lists only the most significant differences. Some of the differences are compatible and are included to indicate that these apparent differences are actually compatible with the specification. This list does not include possible differences due to ambiguity in the specification. This list also omits the SCL extensions to Common Lisp.

The sections in this chapter are arranged in the same order as are those in the *CLtL*.

CLtL: Chapter 2 - Data Types

All atoms (non-lists) that are not symbols are self-evaluating, although the *CLtL* only requires that bit-vectors, numbers, characters, and strings be so.

This difference is compatible.

CLtL: Section 2.2.5 - String Characters

The type string is implemented to be a subtype of the type common. The type string-char is not a subtype of the type common.

CLtL: Section 4.8 - Type Conversion Function

Some uses of **coerce** function signal errors. This is not an incompatibility with the *CLtL*; the following examples are provided only for clarification.

Some examples that signal errors and the reasons for the errors are shown below.

The following is an error because the length is specified.

```
(coerce '(1 2 3) '(vector t 3))
```

The following is an error because the length is specified and does not match the number of elements for *object.*

```
(coerce '(1 2 3) '(vector t 4))
```

The following is an error because *object* is not an element of *result-type.*

(coerce #\c-A 'string-char)

The following is an error because a range is specified.

(coerce 22/7 '(float 0 10))

CLtL: Section 5.1.3 - Special Forms

The Symbolics implementation of Common Lisp does not provide some of the equivalent macro definitions for special forms described in *CLtL* as macros; instead they are implemented as special forms.

This difference is incompatible.

CLtL: Section 5.2.2 - Lambda-Expressions

The arguments for **&rest** parameters have dynamic extent. Furthermore, these arguments should not be modified with the **rplaca** or **rplacd** functions. See the lambda list keyword **&rest** in *Symbolics Common Lisp: Language Dictionary.*

CLtL: Section 7.1.1 - Reference

The **function** special form can return objects that are not functions; for example, **function** can return the contents of the function cell of a macro.

CLtL: Chapter 9 - Declarations

declare forms at the top level are handled according to the Zetalisp rules rather than following the Common Lisp rules that they are an error. Declarations that are embedded inside a form, where allowed by Common Lisp, are evaluated according to Zetalisp rules. See the special form **declare** in *Symbolics Common Lisp: Language Dictionary.*

special declarations embedded inside a form are usually compatible with Common Lisp. See the special form **special** in *Program Development Utilities.*

CLtL: Section 9.1 - Declaration Syntax

The compiler ignores the *value-type* argument in the **the** special form. This difference is compatible.

The **proclaim** function is implemented so that its forms at the top level are evaluated unconditionally at compile time. The CLtL does not state specifically how this evaluation should be done.

CLtL: Section 9.2 - Declaration Specifiers

The declarations **type**, **ftype**, and **optimize** are not implemented. In general all Common Lisp declarations other than **special** are ignored.

This is a minor incompatible difference.

CLtL: Section 9.3 - Type Declaration for Forms

The *value-type* argument in **the** is ignored by the compiler.

This difference is compatible.

CLtL: Section 11.3 - Translating Strings to Symbols

Package-name lookup is not case-sensitive.

This difference is incompatible.

CLtL: Section 13.2 - Predicates on Characters

char-equal does not ignore bits. For example:

```
(char-equal #\A #\Control-A) => NIL
```

Note that Common Lisp specifies that **char-equal** should ignore bits. This difference is incompatible. However, it is likely that the Common Lisp specification might change in the future so that **char-equal** should not ignore bits.

CLtL: Section 14.3 - Modifying Sequences

The **substitute, substitute-if,** and **substitute-if-not** functions are not optimized for detecting the case in which they can return just their argument. This difference is compatible.

delete-duplicates supports the use of the **:replace** keyword. In addition to removing duplicates from the front of the list, the element that stays is moved up to the position of the element that is deleted. **:replace** is not meaningful if **:from-end t** is also used. This is an extension to Common Lisp.

CLtL: Section 14.4 - Searching Sequences for Items

The function **mismatch** erroneously returns **nil** in some cases.

CLtL: Chapter 14 - Sequences, Chapter 15 - Lists

All sequence and list functions that take a two-argument predicate (such as **:test** and **:not-test**) always keep the order of arguments to the predicate consistent with the order of arguments to the sequence or list function. Thus, when there are two sequences and the predicate is called with one item of each, the first argument to the predicate is an element of the first sequence. When there is an item and a sequence, the first argument to the predicate is the item. When there is one sequence and two elements of it are compared, they are always compared in the order they appear in the sequence.

This is not a discrepancy from the *CLtL*; this information is provided for clarification.

CLtL: Section 19.6 - By-position Constructor Functions

The constructor does not evaluate **defstruct** slot initializations in the appropriate lexical environment.

CLtL: Section 20.2 - The Top-Level Loop

A top-level form that returns no values does not set the variable *. The variable * remains unchanged.

This difference is incompatible.

CLtL: Section 21.2 - Creating New Streams

The function **make-echo-stream** is not implemented.

This difference is incompatible.

CLtL: Section 22.1.2 - Parsing of Numbers and Symbols

Setting the value of ***read-base*** greater than 10 causes tokens to fail to be interpreted as numbers rather than symbols. For example, if ***read-base*** is set to 16 (hexadecimal radix), variables with names such as **a**, **b**, and **face** are interpreted as symbols rather than numbers. You can set the values of the variables ***read-extended-ibase-signed-numbers*** and ***read-extended-ibase-unsigned-numbers*** to **t** to cause the tokens to always be interpreted as a number. This difference is incompatible.

The **set-syntax-from-char** function can copy most character attributes rather than being limited to the standard character syntax types shown in Table 22-1, Standard Character Syntax Types.

The Symbolics implementation of Common Lisp does not implement the requirements in Table 22-3, Standard Constituent Character Attributes, about *illegal* character attributes. Changing the syntactic type of space, tab, backspace, newline (also called return), linefeed, page, or rubout to *constituent* or *non-terminating macro* type does not signal an error.

CLtL: Section 22.1.3 - Macro Characters

#p is used for printing pathnames and is followed by a string in double quotes.

This difference is compatible.

CLtL: Section 22.1.4 - Standard Dispatching Macro Character Syntax

Symbols in the ***features*** list must be keywords for the reader macros #+ and #- to work with them. The #+ and #- reader macros read the *feature* that follows them in the keyword package, not in the package that is currently in effect.

CLtL: Section 22.1.5 - The Readtable

The **get-macro-character** function returns **nil** for built-in macros, as they are not defined with **set-macro-character**. This causes the example for the **read-delimited-list** function in the *CLtL* under **read-delimited-list** not to work.

This difference is incompatible.

CLtL: Section 22.1.6 - What the Print Function Produces

The Symbolics implementation of Common Lisp uses the Zetalisp names (the names on the keyboard), rather than the names shown in the *CLtL*, for the printing of characters. This is not completely compatible with Common Lisp.

Slashification is controlled by which tokens the reader interprets as numbers. Only symbols whose printed representations are actual numbers get slashified on printing. A symbol whose printed representation is a potential number and not an actual number does not get slashified. Potential numbers are described in Section 22.1.2, *Parsing of Numbers and Symbols*, in the *CLtL*. This difference is incompatible.

CLtL: Section 22.2.1 - Input from Character Streams

The **read-char** function echoes the character read from the input stream if it is the terminal.

The second value returned by **read-from-string** is at most the length of the string; it is never one greater than the length of the string.

These differences are compatible.

CLtL: Section 22.3.3 - Formatted Output to Character Streams

The ~T directive does not know the column position when the output is directed to a file.

CLtL: Section 23.1.1 - Pathnames

Pathname components of **:unspecific** for the device, directory, type, and version components are allowed in some circumstances.

Pathname hosts are instances; they are not strings or lists of strings. The host component of a pathname should be considered to be a structured component.

CLtL: Section 23.1.2 - Pathname Functions

The **parse-namestring** function uses the Zetalisp rules for **fs:parse-pathname** regarding what a non-null host means, rather than the rules shown in the *CLtL*. Thus, when *host* is not **nil**, *thing* should not contain a host name. This difference is compatible.

The **:junk-allowed** keyword for **parse-namestring** is not implemented to accept **t** as an argument. This difference is incompatible.

CLtL: Section 23. - Loading Files

load uses ***load-pathname-defaults*** as the default for *filename*, rather than using ***default-pathname-defaults***. ***load-pathname-defaults*** is a Zetalisp defaults-alist, whereas ***default-pathname-defaults*** is a Common Lisp default pathname.

CLtL: Section 23.5 - Accessing Directories

The **directory** function returns **nil** if no files matching *pathname* are found, but still signals an error for other file lookup errors, such as not finding the directory.

This difference is compatible.

CLtL: Section 25.1 - The Compiler

The **compile-file** function accepts the keyword **:load**. This is provided for compatibility with Spice Lisp.

This difference is compatible.

CLtL: Section 25.3 - Debugging Tools

The **describe** function is the Zetalisp **describe**; it returns its argument after describing the object. This difference is incompatible.

The function **dribble** calls **zl:dribble-start** and **zl:dribble-end**. This means that **dribble** does not return until the dribbling has been completed, because it creates a new command loop to do the dribbling. This difference is compatible.

CLtL: Section 25.4.1 - Time Functions

The *time-zone* argument for the **decode-universal-time** function currently suppresses checking for daylight savings time, just as the **encode-universal-time** function does. It is not clear whether this difference actually constitutes a discrepancy from the *CLtL* specification; this is being checked. However, the compatibility note in the *CLtL* about *time-zone* in Zetalisp is obsolete. **get-decoded-time** now returns *time-zone*.

CLtL: Section 25.4.2 - Other Environment Inquiries

Symbols in the ***features*** list must be keywords for the reader macros #+ and #- to work with them. The #+ and #- reader macros read the *feature* that follows them in the keyword package, not in the package that is currently in effect.

26. Symbols in Package Common Lisp Documented in Volumes 2a and 2b

* (Function)
+ (Function)
- (Function)
/ (Function)
/= (Function)
1- (Function)
1+ (Function)
< (Function)
<= (Function)
= (Function)
> (Function)
>= (Function)
abs (Function)
acons (Function)
acos (Function)
acosh (Function)
adjoin (Function)
adjustable-array-p (Function)
adjust-array (Function)
&allow-other-keys (Lambda List Keyword)
alpha-char-p (Function)
alphanumericp (Function)
and (Special Form, Type Specifier)
append (Function)
aref (Function)
array (Type Specifier)
array-dimension (Function)
array-dimension-limit (Constant)
array-dimensions (Function)
array-element-type (Function)
array-has-fill-pointer-p (Function)
array-in-bounds-p (Function)
arrayp (Function)
array-rank (Function)
array-rank-limit (Constant)
array-row-major-index (Function)
array-total-size (Function)
array-total-size-limit (Constant)

ash (Function)
asin (Function)
asinh (Function)
assert (Macro)
assoc (Function)
assoc-if (Function)
assoc-if-not (Function)
atan (Function)
atanh (Function)
atom (Function, Type Specifier)
&aux (Lambda List Keyword)
bignum (Type Specifier)
bit (Function, Type Specifier)
bit-and (Function)
bit-andc1 (Function)
bit-andc2 (Function)
bit-eqv (Function)
bit-ior (Function)
bit-nand (Function)
bit-nor (Function)
bit-not (Function)
bit-orc1 (Function)
bit-orc2 (Function)
bit-vector (Type Specifier)
bit-vector-p (Function)
bit-xor (Function)
block (Special Form)
&body (Lambda List Keyword)
boole (Function)
boole-1 (Constant)
boole-2 (Constant)
boole-and (Constant)
boole-andc1 (Constant)
boole-andc2 (Constant)
boole-c1 (Constant)
boole-c2 (Constant)
boole-clr (Constant)
boole-eqv (Constant)
boole-ior (Constant)
boole-nand (Constant)
boole-nor (Constant)
boole-orc1 (Constant)
boole-orc2 (Constant)

boole-set (Constant)
boole-xor (Constant)
both-case-p (Function)
boundp (Function)
break-on-warnings (Variable)
butlast (Function)
byte (Function)
byte-position (Function)
byte-size (Function)
caaaar (Function)
caaadr (Function)
caaar (Function)
caadar (Function)
caaddr (Function)
caadr (Function)
caar (Function)
cadaar (Function)
cadadr (Function)
cadar (Function)
caddar (Function)
cadddr (Function)
caddr (Function)
cadr (Function)
car (Function)
case (Special Form)
catch (Special Form)
ccase (Special Form)
cdaaar (Function)
cdaadr (Function)
cdaar (Function)
cdadar (Function)
cdaddr (Function)
cdadr (Function)
cdar (Function)
cddaar (Function)
cddadr (Function)
cddar (Function)
cdddar (Function)
cddddr (Function)
cdddr (Function)
cddr (Function)
cdr (Function)
ceiling (Function)

cerror (Function)
char (Function)
char/= (Function)
char< (Function)
char<= (Function)
char= (Function)
char> (Function)
char>= (Function)
character (Function, Type Specifier)
characterp (Function)
char-bit (Function)
char-bits (Function)
char-bits-limit (Constant)
char-code (Function)
char-code-limit (Constant)
char-control-bit (Constant)
char-downcase (Function)
char-equal (Function)
char-font (Function)
char-font-limit (Constant)
char-greaterp (Function)
char-hyper-bit (Constant)
char-int (Function)
char-lessp (Function)
char-meta-bit (Constant)
char-name (Function)
char-not-equal (Function)
char-not-greaterp (Function)
char-not-lessp (Function)
char-super-bit (Constant)
char-upcase (Function)
check-type (Macro)
cis (Function)
clrhash (Function)
code-char (Function)
coerce (Function)
common (Type Specifier)
commonp (Function)
compiled-function (Type Specifier)
compiled-function-p (Function)
compiler-let (Special Form)
complex (Function, Type Specifier)
complexp (Function)

concatenate (Function)
cond (Special Form)
conjugate (Function)
cons (Function, Type Specifier)
user::constant-p (Function)
copy-alist (Function)
copy-list (Function)
copy-seq (Function)
copy-symbol (Function)
copy-tree (Function)
cos (Function)
cosh (Function)
count (Function)
count-if (Function)
count-if-not (Function)
ctypecase (Special Form)
decf (Macro)
declare (Special Form)
decode-float (Function)
defconstant (Special Form)
define-modify-macro (Macro)
define-setf-method (Macro)
defmacro (Macro, Subsection)
defparameter (Special Form)
defsetf (Macro)
defstruct (Macro)
defun (Special Form)
defvar (Special Form)
delete (Function)
delete-duplicates (Function)
delete-if (Function)
delete-if-not (Function)
denominator (Function)
deposit-field (Function)
digit-char (Function)
digit-char-p (Function)
do (Special Form)
do* (Special Form)
do-all-symbols (Special Form)
documentation (Function)
do-external-symbols (Special Form)
dolist (Special Form)
do-symbols (Special Form)

dotimes (Special Form)
double-float (Type Specifier)
double-float-epsilon (Constant)
double-float-negative-epsilon (Constant)
dpb (Function)
ecase (Special Form)
eighth (Function)
elt (Function)
endp (Function)
&environment (Lambda List Keyword)
eq (Function)
eql (Function)
equal (Function)
equalp (Function)
error (Flavor, Function)
etypecase (Special Form)
eval (Function)
evenp (Function)
every (Function)
exp (Function)
export (Function)
expt (Function)
fboundp (Function)
fceiling (Function)
ffloor (Function)
fifth (Function)
fill (Function)
fill-pointer (Function)
find (Function)
find-all-symbols (Function)
find-if (Function)
find-if-not (Function)
find-package (Function)
find-symbol (Function)
first (Function)
fixnum (Type Specifier)
flet (Special Form)
float (Function, Type Specifier)
float-digits (Function)
floatp (Function)
float-precision (Function)
float-radix (Function)
float-sign (Function)

floor (Function)
fmakunbound (Function)
fourth (Function)
fround (Function)
ftruncate (Function)
funcall (Function)
function (Special Form, Type Specifier)
functionp (Function)
gcd (Function)
gensym (Function)
gentemp (Function)
get (Function)
getf (Function)
gethash (Function)
get-properties (Function)
get-setf-method (Function)
go (Special Form)
graphic-char-p (Function)
hash-table (Type Specifier)
hash-table-count (Function)
hash-table-p (Function)
identity (Function)
if (Special Form)
ignore (Function)
imagpart (Function)
import (Function)
incf (Macro)
inline (Declaration)
in-package (Function)
int-char (Function)
integer (Type Specifier)
integer-decode-float (Function)
integer-length (Function)
integerp (Function)
intern (Function)
intersection (Function)
isqrt (Function)
&key (Lambda List Keyword)
keyword (Type Specifier)
keywordp (Function)
labels (Special Form)
lambda (Special Form)
lambda-list-keywords (Variable)

lambda-parameters-limit (Constant)
last (Function)
lcm (Function)
ldb (Function)
ldb-test (Function)
ldiff (Function)
least-negative-double-float (Constant)
least-negative-long-float (Constant)
least-negative-short-float (Constant)
least-negative-single-float (Constant)
least-positive-double-float (Constant)
least-positive-long-float (Constant)
least-positive-short-float (Constant)
least-positive-single-float (Constant)
length (Function)
let (Special Form)
let* (Special Form)
list (Function, Type Specifier)
list* (Function)
list-all-packages (Function)
list-length (Function)
listp (Function)
locally (Macro)
log (Function)
logand (Function)
logandc1 (Function)
logandc2 (Function)
logbitp (Function)
logcount (Function)
logeqv (Function)
logior (Function)
lognand (Function)
lognor (Function)
lognot (Function)
logorc1 (Function)
logorc2 (Function)
logtest (Function)
logxor (Function)
long-float (Type Specifier)
long-float-epsilon (Constant)
long-float-negative-epsilon (Constant)
loop (Macro)
lower-case-p (Function)

macroexpand (Function)
macroexpand-1 (Function)
macro-function (Function)
macroexpand-hook (Variable)
macrolet (Special Form)
make-array (Function)
make-char (Function)
make-hash-table (Function)
make-list (Function)
make-package (Function)
make-random-state (Function)
make-sequence (Function)
make-string (Function)
make-symbol (Function)
makunbound (Function)
map (Function)
mapc (Function)
mapcan (Function)
mapcar (Function)
mapcon (Function)
maphash (Function)
mapl (Function)
maplist (Function)
mask-field (Function)
max (Function)
member (Function, Type Specifier)
member-if (Function)
member-if-not (Function)
merge (Function)
min (Function)
minusp (Function)
mismatch (Function)
mod (Function, Type Specifier)
modules (Variable)
most-negative-double-float (Constant)
most-negative-fixnum (Constant)
most-negative-long-float (Constant)
most-negative-short-float (Constant)
most-negative-single-float (Constant)
most-positive-double-float (Constant)
most-positive-fixnum (Constant)
most-positive-long-float (Constant)
most-positive-short-float (Constant)

most-positive-single-float (Constant)
multiple-value-bind (Special Form)
multiple-value-call (Special Form)
multiple-value-list (Special Form)
multiple-value-prog1 (Special Form)
name-char (Function)
nbutlast (Function)
nconc (Function)
nintersection (Function)
ninth (Function)
not (Function, Type Specifier)
notany (Function)
notevery (Function)
notinline (Declaration)
nreconc (Function)
nreverse (Function)
nset-difference (Function)
nset-exclusive-or (Function)
nstring-capitalize (Function)
nstring-downcase (Function)
nstring-upcase (Function)
nsublis (Function)
nsubst (Function)
nsubst-if (Function)
nsubst-if-not (Function)
nsubstitute (Function)
nsubstitute-if (Function)
nsubstitute-if-not (Function)
nth (Function)
nthcdr (Function)
null (Function, Type Specifier)
number (Type Specifier)
numberp (Function)
numerator (Function)
nunion (Function)
oddp (Function)
&optional (Lambda List Keyword)
or (Special Form, Type Specifier)
package (Type Specifier)
package (Variable)
package-name (Function)
package-nicknames (Function)
package-shadowing-symbols (Function)

package-used-by-list (Function)
package-use-list (Function)
pairlis (Function)
pathname (Type Specifier)
phase (Function)
pi (Constant)
plusp (Function)
pop (Function)
position (Function)
position-if (Function)
position-if-not (Function)
print-base (Variable)
print-radix (Variable)
proclaim (Function)
prog (Special Form)
prog* (Special Form)
prog1 (Special Form)
prog2 (Special Form)
progn (Special Form)
provide (Function)
psetf (Macro)
psetq (Macro)
push (Function)
pushnew (Function)
quote (Special Form)
random (Function)
random-state (Type Specifier)
random-state (Variable)
random-state-p (Function)
rassoc (Function)
rassoc-if (Function)
rassoc-if-not (Function)
ratio (Type Specifier)
rational (Function, Type Specifier)
rationalize (Function)
rationalp (Function)
read-base (Variable)
read-default-float-format (Variable)
readtable (Type Specifier)
realpart (Function)
reduce (Function)
rem (Function)
remf (Macro)

remhash (Function)
remove (Function)
remove-duplicates (Function)
remove-if (Function)
remove-if-not (Function)
remprop (Function)
rename-package (Function)
replace (Function)
require (Function)
&rest (Lambda List Keyword)
rest (Function)
return (Special Form)
return-from (Special Form)
revappend (Function)
reverse (Function)
rotatef (Macro)
round (Function)
rplaca (Function)
rplacd (Function)
satisfies (Type Specifier)
sbit (Function)
scale-float (Function)
schar (Function)
search (Function)
second (Function)
sequence (Type Specifier)
set (Function)
set-char-bit (Function)
set-difference (Function)
set-exclusive-or (Function)
setf (Macro)
setq (Special Form)
seventh (Function)
shadow (Function)
shadowing-import (Function)
shiftf (Macro)
short-float (Type Specifier)
short-float-epsilon (Constant)
short-float-negative-epsilon (Constant)
signed-byte (Type Specifier)
signum (Function)
simple-array (Type Specifier)
simple-bit-vector (Type Specifier)

simple-bit-vector-p (Function)
simple-string (Type Specifier)
simple-string-p (Function)
simple-vector (Type Specifier)
simple-vector-p (Function)
sin (Function)
single-float (Type Specifier)
single-float-epsilon (Constant)
single-float-negative-epsilon (Constant)
sinh (Function)
sixth (Function)
some (Function)
sort (Function)
special-form-p (Function)
sqrt (Function)
stable-sort (Function)
standard-char (Type Specifier)
standard-char-p (Function)
stream (Type Specifier)
string (Function, Type Specifier)
string/= (Function)
string< (Function)
string<= (Function)
string= (Function)
string> (Function)
string>= (Function)
string-capitalize (Function)
string-char (Type Specifier)
string-char-p (Function)
string-downcase (Function)
string-equal (Function)
string-greaterp (Function)
string-left-trim (Function)
string-lessp (Function)
string-not-equal (Function)
string-not-greaterp (Function)
string-not-lessp (Function)
stringp (Function)
string-right-trim (Function)
string-trim (Function)
string-upcase (Function)
structure (Type Specifier)
sublis (Function)

subseq (Function)
subsetp (Function)
subst (Function)
subst-if (Function)
subst-if-not (Function)
substitute (Function)
substitute-if (Function)
substitute-if-not (Function)
subtypep (Function)
svref (Function)
sxhash (Function)
symbol (Type Specifier)
symbol-function (Function)
symbol-name (Function)
symbolp (Function)
symbol-package (Function)
symbol-plist (Function)
symbol-value (Function)
t (Type Specifier)
tagbody (Special Form)
tailp (Function)
tan (Function)
tanh (Function)
tenth (Function)
the (Special Form)
third (Function)
throw (Special Form)
tree-equal (Function)
truncate (Function)
typecase (Special Form)
type-of (Function)
typep (Function)
unexport (Function)
unintern (Function)
union (Function)
unless (Macro)
unsigned-byte (Type Specifier)
unuse-package (Function)
unwind-protect (Special Form)
upper-case-p (Function)
use-package (Function)
values (Function, Type Specifier)
values-list (Function)

vector (Function, Type Specifier)
vectorp (Function)
vector-pop (Function)
vector-push (Function)
vector-push-extend (Function)
warn (Function)
when (Macro)
&whole (Lambda List Keyword)
zerop (Function)

27. Symbols in Package Symbolics Common Lisp Documented in Volumes 2a and 2b

≠ (Function)
≤ (Function)
≥ (Function)
2d-array-blt (Function)
alphalessp (Function)
arglist (Function)
args-info (Function)
array-has-leader-p (Function)
array-leader (Function)
array-leader-length (Function)
array-leader-length-limit (Constant)
ascii-code (Function)
ascii-to-char (Function)
ascii-to-string (Function)
bitblt (Function)
boundp-in-closure (Function)
breakon (Function)
catch-error (Function)
catch-error-restart (Special Form)
catch-error-restart-if (Special Form)
change-instance-flavor (Function)
char≠ (Function)
char≤ (Function)
char≥ (Function)
char-fat-p (Function)
char-flipcase (Function)
char-mouse-button (Function)
char-mouse-equal (Function)
char-to-ascii (Function)
circular-list (Function)
closure-function (Function)
compile-flavor-methods (Macro)
cond-every (Special Form)
condition (Flavor)
condition-bind (Special Form)
condition-bind-default (Special Form)
condition-bind-default-if (Special Form)

condition-bind-if (Special Form)
condition-call (Special Form)
condition-call-if (Special Form)
condition-case (Special Form)
condition-case-if (Special Form)
cons-in-area (Function)
continue-whopper (Function)
copy-array-contents (Function)
copy-array-contents-and-leader (Function)
copy-array-portion (Function)
copy-list* (Function)
copy-tree-share (Function)
cosd (Function)
debugging-info (Function)
decode-raster-array (Function)
def (Special Form)
deff (Special Form)
defflavor (Special Form)
deffunction (Special Form)
defgeneric (Special Form)
define-global-handler (Macro)
define-method-combination (Special Form)
define-simple-method-combination (Special Form)
define-symbol-macro (Special Form)
deflambda-macro (Special Form)
defmacro-in-flavor (Special Form)
defmethod (Special Form)
defpackage (Special Form)
defprop (Special Form)
defselect (Special Form)
defstruct-define-type (Macro)
defsubst (Special Form)
defsubst-in-flavor (Special Form)
defun-in-flavor (Special Form)
defwhopper (Special Form)
defwhopper-subst (Macro)
defwrapper (Macro)
deposit-byte (Function)
describe-defstruct (Function)
describe-function (Function)
describe-package (Function)
destructuring-bind (Special Form)
do-local-symbols (Special Form)

equal-typep (Function)
errorp (Function)
error-restart (Special Form)
error-restart-loop (Special Form)
false (Function)
fdefine (Function)
fdefinedp (Function)
fdefinition (Function)
flavor-allows-init-keyword-p (Function)
fundefine (Function)
get-handler-for (Generic Function)
globalize (Function)
g-l-p (Function)
ignore-errors (Special Form)
instance (Type Specifier)
intern-local (Function)
intern-local-soft (Function)
intern-soft (Function)
lambda-macro (Special Form)
letf (Special Form)
letf* (Special Form)
let-globally (Special Form)
let-globally-if (Special Form)
let-if (Special Form)
lexpr-continue-whopper (Function)
lexpr-send (Function)
lexpr-send-if-handles (Function)
list*-in-area (Function)
list-array-leader (Function)
list-in-area (Function)
load-byte (Function)
location-boundp (Function)
location-contents (Function)
location-makunbound (Function)
locative (Type Specifier)
locativep (Function)
locf (Macro)
loop-finish (Macro)
lsh (Function)
macro (Special Form)
make-character (Function)
make-condition (Function)
make-heap (Function)

make-instance (Generic Function)
make-mouse-char (Function)
make-plane (Function)
make-raster-array (Function)
makunbound-globally (Function)
makunbound-in-closure (Function)
mexp (Function)
modify-hash (Function)
mouse-char-p (Function)
named-structure-invoke (Function)
named-structure-p (Function)
named-structure-symbol (Function)
ncons (Function)
ncons-in-area (Function)
neq (Function)
nleft (Function)
nlistp (Function)
nstring-capitalize-words (Function)
nsubstring (Function)
nsymbolp (Function)
once-only (Macro)
operation-handled-p (Function)
package-external-symbols (Function)
pkg-add-relative-name (Function)
pkg-delete-relative-name (Function)
pkg-find-package (Function)
pkg-kill (Function)
plane-aref (Function)
plane-default (Function)
plane-extension (Function)
progw (Special Form)
raster-aref (Function)
raster-index-offset (Function)
raster-width-and-height-to-make-array-dimensions (Function)
recompile-flavor (Function)
record-source-file-name (Function)
rot (Function)
select (Special Form)
selector (Special Form)
selectq-every (Special Form)
self (Variable)
send (Function)
send-if-handles (Function)

signal (Function)
signal-proceed-case (Special Form)
sind (Function)
sort-grouped-array (Function)
sort-grouped-array-group-key (Function)
string≠ (Function)
string≤ (Function)
string≥ (Function)
string-append (Function)
string-capitalize-words (Function)
string-compare (Function)
string-exact-compare (Function)
string-fat-p (Function)
string-flipcase (Function)
string-length (Function)
string-nconc (Function)
string-nconc-portion (Function)
string-nreverse (Function)
string-pluralize (Function)
string-reverse (Function)
string-search (Function)
string-search-char (Function)
string-search-exact (Function)
string-search-exact-char (Function)
string-search-not-char (Function)
string-search-not-exact-char (Function)
string-search-not-set (Function)
string-search-set (Function)
string-to-ascii (Function)
substring (Function)
swaphash (Function)
symbol-value-globally (Function)
symbol-value-in-closure (Function)
symbol-value-in-instance (Function)
table-size (Function)
tand (Function)
true (Function)
unbreakon (Function)
undefine-global-handler (Macro)
undefun (Function)
unwind-protect-case (Macro)
variable-boundp (Special Form)
variable-makunbound (Special Form)

vector-push-portion-extend (Function)
where-is (Function)
without-floating-underflow-traps (Special Form)
xcons (Function)
xcons-in-area (Function)

28. Symbols in Package Global Documented in Volumes 2a and 2b

zl:*$ (Function)
zl:+$ (Function)
zl:-$ (Function)
zl:/ (Function)
zl:/$ (Function)
zl:1-$ (Function)
zl:1+$ (Function)
sys:%32-bit-difference (Function)
sys:%32-bit-plus (Function)
zl:@define (Macro)
zl:\\ (Function)
zl:\\\\ (Function)
zl:^ (Function)
zl:^$ (Function)
zl:add1 (Function)
zl:adjust-array-size (Function)
sys:*all-flavor-names* (Variable)
zl:aloc (Function)
zl:ap-1 (Function)
zl:ap-2 (Function)
zl:ap-leader (Function)
zl:apply (Function)
zl:ar-1 (Function)
zl:ar-2 (Function)
zl:arg (Function)
sys:%args-info (Function)
zl:argument-typecase (Special Form)
zl:array (Macro)
zl:*array (Function)
zl:array-#-dims (Function)
zl:array-active-length (Function)
sys:array-bits-per-element (Function, Variable)
zl:array-dimension-n (Function)
zl:arraydims (Function)
sys:array-displaced-p (Function)
sys:array-element-size (Function)
sys:array-elements-per-q (Function, Variable)
zl:array-grow (Function)

sys:array-indexed-p (Function)
sys:array-indirect-p (Function)
zl:array-length (Function)
zl:array-pop (Function)
zl:array-push (Function)
zl:array-push-extend (Function)
zl:array-push-portion-extend (Function)
sys:array-type (Function)
sys:array-types (Function)
zl:as-1 (Function)
zl:as-2 (Function)
zl:ascii (Function)
zl:aset (Function)
zl:ass (Function)
zl:assoc (Function)
zl:assq (Function)
zl:atan (Function)
zl:atan2 (Function)
zl:base (Variable)
zl:bigp (Function)
zl:bit-test (Function)
zl:break (Flavor)
zl:call (Function)
zl:car-location (Function)
zl:caseq (Special Form)
zl:*catch (Special Form)
sys:char-subindex (Function)
zl:check-arg (Macro)
zl:check-arg-type (Macro)
zl:closure (Function)
zl:closure-alist (Function)
zl:closurep (Function)
zl:closure-variables (Function)
zl:clrhash-equal (Function)
zl:comment (Special Form)
zl:copyalist (Function)
zl:copy-closure (Function)
zl:copylist (Function)
zl:copylist* (Function)
zl:copysymbol (Function)
zl:copytree (Function)
zl:copytree-share (Function)
zl:defconst (Special Form)

zl:deflambda-macro-displace (Special Form)
zl:defmacro-displace (Macro)
zl:defstruct (Macro)
zl:defunp (Macro)
sys:defvar-resettable (Special Form)
sys:defvar-standard (Special Form)
zl:del (Function)
zl:delete (Function)
zl:del-if (Function)
zl:del-if-not (Function)
zl:delq (Function)
zl:desetq (Special Form)
zl:dfloat (Function)
zl:difference (Function)
zl:dispatch (Special Form)
zl:displace (Function)
zl:dlet (Special Form)
zl:dlet* (Special Form)
zl:do*-named (Special Form)
zl:dolist (Special Form)
zl:do-named (Special Form)
zl:dotimes (Special Form)
zl:equal (Function)
zl:every (Function)
zl:expt (Function)
zl:ferror (Flavor, Function)
zl:fillarray (Function)
zl:find-position-in-list (Function)
zl:find-position-in-list-equal (Function)
zl:firstn (Function)
zl:fix (Function)
zl:fixnump (Function)
zl:fixp (Function)
zl:fixr (Function)
zl:float (Function)
zl:flonump (Function)
zl:fset (Function)
zl:fset-carefully (Function)
zl:fsignal (Function)
zl:fsymeval (Function)
sys:function-cell-location (Function)
zl:gcd (Function)
zl:gensym (Function)

zl:get (Function)
zl:getchar (Function)
zl:getcharn (Function)
zl:get-flavor-handler-for (Function)
zl:gethash (Function)
zl:gethash-equal (Function)
zl:getl (Function)
zl:get-pname (Function)
zl:greaterp (Function)
zl:haipart (Function)
zl:haulong (Function)
zl:ibase (Variable)
zl:implode (Function)
zl:intern (Function)
zl:intersection (Function)
zl:keyword-extract (Special Form)
zl:length (Function)
zl:lessp (Function)
zl:let-closed (Special Form)
zl:lexpr-funcall (Function)
zl:listarray (Function)
zl:listify (Function)
zl:listp (Function)
sys:local-declarations (Variable)
zl:local-declare (Special Form)
zl:locate-in-closure (Function)
zl:locate-in-instance (Function)
zl:log (Function)
zl:logand (Function)
sys:%logdpb (Function)
zl:logior (Function)
sys:%logldb (Function)
zl:logxor (Function)
zl:loop (Macro)
zl:make-array (Function)
zl:make-array-into-named-structure (Function)
zl:make-equal-hash-table (Function)
zl:make-hash-table (Function)
zl:make-list (Function)
zl:make-raster-array (Function)
zl:maknam (Function)
zl:map (Function)
zl:mapatoms (Function)

zl:mapatoms-all (Function)
zl:maphash-equal (Function)
zl:mem (Function)
zl:memass (Function)
zl:member (Function)
zl:memq (Function)
zl:minus (Function)
zl:multiple-value (Special Form)
zl:nintersection (Function)
zl:nlistp (Function)
zl:nreverse (Function)
zl:nsublis (Function)
zl:nsubst (Function)
sys:number-into-array (Function)
zl:nunion (Function)
zl:package (Variable)
sys:package-cell-location (Function)
zl:pairlis (Function)
zl:parse-ferror (Flavor, Function)
zl:parse-number (Function)
zl:pkg-bind (Macro)
zl:pkg-global-package (Variable)
zl:pkg-goto (Function)
sys:pkg-keyword-package (Variable)
zl:pkg-name (Function)
zl:pkg-system-package (Variable)
zl:plane-aset (Function)
zl:plane-origin (Function)
zl:plane-ref (Function)
zl:plane-store (Function)
zl:plist (Function)
zl:plus (Function)
zl:pop (Macro)
zl:prinlength (Variable)
zl:prinlevel (Variable)
zl:progv (Special Form)
sys:property-cell-location (Function)
zl:psetq (Special Form)
zl:push (Macro)
zl:push-in-area (Macro)
zl:puthash (Function)
zl:puthash-equal (Function)
zl:putprop (Function)

zl:quotient (Function)
zl:random (Function)
zl:rass (Function)
zl:rassoc (Function)
zl:rassq (Function)
zl:rational (Function)
zl:rationalp (Function)
zl:rem (Function)
zl:remainder (Function)
zl:remhash-equal (Function)
zl:rem-if (Function)
zl:rem-if-not (Function)
zl:remob (Function)
zl:remove (Function)
zl:remprop (Function)
zl:remq (Function)
zl:rest1 (Function)
zl:rest2 (Function)
zl:rest3 (Function)
zl:rest4 (Function)
sys:return-array (Function)
zl:return-list (Function)
zl:reverse (Function)
zl:samepnamep (Function)
zl:sassoc (Function)
zl:sassq (Function)
zl:selectq (Special Form)
zl:setarg (Function)
zl:setf (Macro)
zl:set-globally (Function)
zl:set-in-closure (Function)
zl:set-in-instance (Function)
zl:setplist (Function)
zl:setq-standard-value (Special Form)
zl:signp (Special Form)
zl:some (Function)
zl:sort (Function)
zl:sortcar (Function)
zl:sqrt (Function)
zl:stable-sort (Function)
zl:stable-sortcar (Function)
sys:standard-value-let (Macro)
sys:standard-value-let* (Macro)

sys:standard-value-progv (Macro)
zl:store-array-leader (Function)
zl:string≠ (Function)
zl:string≤ (Function)
zl:string≥ (Function)
zl:string< (Function)
sys:%string= (Function)
zl:string= (Function)
zl:string> (Function)
zl:string-capitalize-words (Function)
zl:string-compare (Function)
zl:string-downcase (Function)
sys:%string-equal (Function)
zl:string-equal (Function)
zl:string-exact-compare (Function)
zl:string-flipcase (Function)
zl:string-greaterp (Function)
zl:string-left-trim (Function)
zl:string-lessp (Function)
zl:string-nconc (Function)
zl:string-not-equal (Function)
zl:string-not-greaterp (Function)
zl:string-not-lessp (Function)
zl:string-nreverse (Function)
zl:string-pluralize (Function)
zl:string-reverse (Function)
zl:string-reverse-search (Function)
zl:string-reverse-search-char (Function)
zl:string-reverse-search-exact (Function)
zl:string-reverse-search-exact-char (Function)
zl:string-reverse-search-not-char (Function)
zl:string-reverse-search-not-exact-char (Function)
zl:string-reverse-search-not-set (Function)
zl:string-reverse-search-set (Function)
zl:string-right-trim (Function)
zl:string-search (Function)
sys:%string-search-char (Function)
zl:string-search-char (Function)
zl:string-search-exact (Function)
sys:%string-search-exact-char (Function)
zl:string-search-exact-char (Function)
zl:string-search-not-char (Function)
zl:string-search-not-exact-char (Function)

zl:string-search-not-set (Function)
zl:string-search-set (Function)
zl:string-trim (Function)
zl:string-upcase (Function)
zl:sub1 (Function)
zl:sublis (Function)
zl:subrp (Function)
zl:subset (Function)
zl:subset-not (Function)
zl:subst (Function)
zl:swapf (Macro)
zl:swaphash-equal (Function)
zl:symeval (Function)
zl:symeval-globally (Function)
zl:symeval-in-closure (Function)
zl:symeval-in-instance (Function)
zl:*throw (Function)
zl:times (Function)
sys:trace-conditions (Variable)
zl:typecase (Special Form)
zl:typep (Function)
zl:union (Function)
zl:value-cell-location (Function)
zl:variable-location (Special Form)

Index