Symbolics Common Lisp Language Concepts

Organization of Symbolics Common Lisp Documentation

Symbolics Common Lisp (SCL) is 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. SCL 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 documentation into several parts, each reflecting a different stage of familiarity with Lisp.

For an overview of Symbolics Common Lisp, see the section "Overview of Symbolics Common Lisp". This section 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.

If you are unfamiliar with the Symbolics notation conventions for Lisp documentation, see the section "Understanding Notation Conventions".

Your reference guide to Symbolics Common Lisp (SCL), the Symbolics implementation of the Lisp language, consists of the following volumes:

Symbolics Common Lisp Language Concepts
: Documents the basic language concepts of Lisp, including data types, type specifiers, functions and dynamic closures, inline functions and macros, evaluation, scoping, flow of control, declarations, and compatibility issues.

Symbolics Common Lisp Programming Constructs
: Documents the higher-level programming constructs of Lisp, including structures, CLOS, Flavors, conditions, packages, and input/output facilities (including the reader, printed representation, input and output functions, and streams).

Symbolics Common Lisp Dictionary
: An alphabetic dictionary of all Lisp objects documented in the previous two volumes.

The first two volumes give a conceptual presentation of Symbolics Common Lisp, and provide in-depth coverage of topics presented in the Overview. The Dictionary is the most detailed part of the documentation. 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 appear in alphabetical order, this volume of Symbolics Common Lisp is not indexed; the other volumes are 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)
```

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".) 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**.

(flavor:method :clear-window tv:sheet) *Method*

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

:tyo *char* *Message*

Puts the *char* into the stream. 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* *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".

(flavor:method :bottom tv:sheet) *bottom-edge* *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".

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".

" 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".

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".

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".

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

Overview of Symbolics Common Lisp

This chapter provides you with a sense of the basic concepts and terms in Symbolics Common Lisp, in a form that you can read at a single sitting. New users should find this material of particular interest.

If you are unfamiliar with the Symbolics notation conventions for Lisp documentation, see the section "Understanding Notation Conventions".

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. For details about the relationship between these dialects, see the section "Lisp Dialects Available in Genera".

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 the documentation and are covered in the context of various other topics. See the section "Cells and Locatives". See the section "Special Forms and Built-in Macros".

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.

Overview of Data Types

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. We say that an object is "of" a datatype if the object is a member of the set that makes up the type.

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*, which 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**.

Subtypes of a common supertype form an *exhaustive union* formed by the supertype if every object belonging to the supertype belongs to at least one of the subtypes. 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 a *partition*. For example, the types **bignum** and **fixnum** form a *partition* of the type **integer**, since every integer is either a **fixnum** or a **bignum** but not both. If all elements of a supertype are members of one of the mutually disjoint subtypes, this forms an *exhaustive partition*.

For a complete list of Symbolics Common Lisp data types, see the section "Hierarchy of Data Types".

Types of 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. The function **type-of** returns the most specific type that can be conveniently computed and is likely to be useful to the user. For example:

```
(type-of 5/7) => RATIO
```

See the section "Type-checking Differences Between Symbolics Common Lisp and Zetalisp".

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)**.

Overview of Numbers

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

- *Rational Numbers* are used for exact mathematical calculations. These include:
  - *Integers* are rational numbers without a fractional part, such as 0, 1, 2.
  - *Ratios* are pairs of integers, representing the numerator and denominator of a number, for example, 15/16, -26/3.
- *Floating-point Numbers* are used for approximate mathematical calculations. Symbolics Common Lisp supports two forms:
- ° *Single-floats* are single-precision floating-point numbers, for example, 1.0e-45.
  - *Double-floats* are double-precision floating-point numbers, for example, 5.0d-324.
- *Complex Numbers* are used to represent the mathematical concept of the same 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 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 5. 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.

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".) 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". 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".

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

These are just a few of the possible operations. The above 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*. See figure ! for an illustration of a single cons cell.

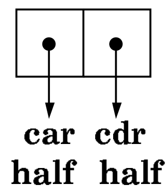


Figure 1. Single Cons Cell

The basic operations on the cons data type are:

cons and **xcons**	Create a cons with a specified car and cdr.
consp	Determines if an object is a cons.
car	Determines the car of the cons.
cdr	Determines 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 lists are constructed 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 denote 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.

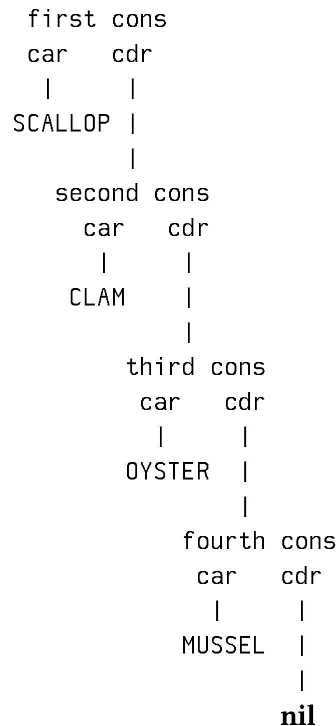
The list of the above example can thus be created by:

```
(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:



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**.

The form of this structure and its recursive generation, make it easy to generate functions to search through lists, extract various parts of lists, and the like.

Special Kinds of Lists

- Property Lists

A table in which each of the items has some property associated with it is called a property list. 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 members 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; they 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*. This term is a misnomer, 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)
```

Conses are the building blocks for a another 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 association list is a list, the elements of which are conses. The conses that compose an association list are not required to be dotted pairs, but they can be. The car of one of these conses is called the *indicator*, and the cdr is called the *value*. The table above is represented as:

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

The same kinds of abstract operations that can be performed on property lists can be performed on association lists, but because of their more complicated structure, additional operations can also be performed on them. You can, for example, search on an indicator through an association list to find a value *or* on a value to find an indicator. The function **pairlis** creates an association list by pairing elements from each of two lists.

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:

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

Figure ! is a diagram of this structure.

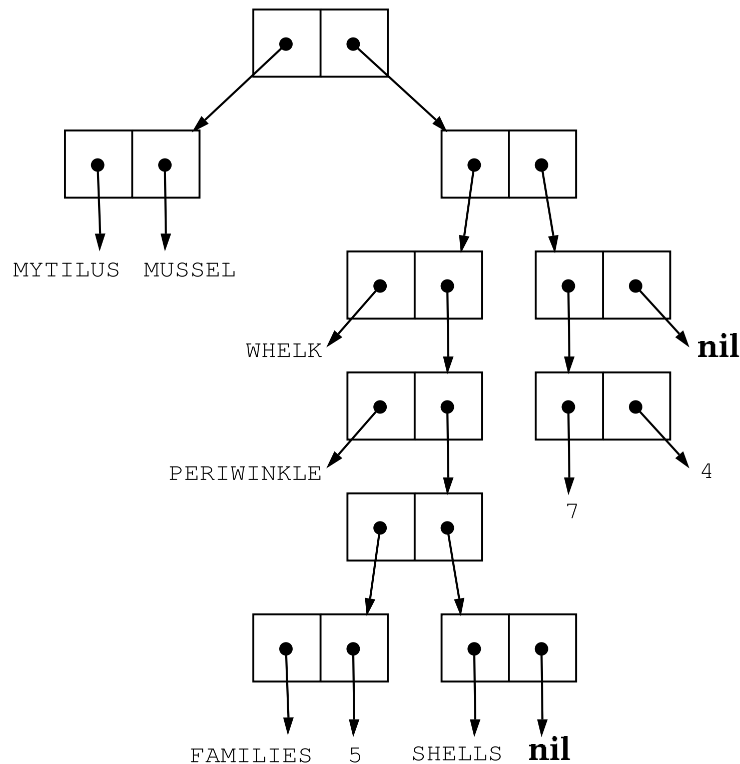


Figure 2. Diagram of a Tree Structure

- Circular Lists

A circular list is a simple list whose last cons's tail points either to the first cons of the list, or another cons in 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 Lists

Symbolics Common Lisp uses a special internal representation called *cdr-coding* 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.

Overview of Arrays

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.

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".

Zetalisp Note: 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".

The basic functions related to arrays enable you to create arrays (**make-array**), access elements (**aref**), and alter elements (**setf** used with **aref**).

There are many types of array operations. Most of these can be done with specialized array functions, while some can be done with more general-purpose sequence functions.

Several advanced and more specialized programming practices are also supported. See the section "Advanced Concepts of Arrays".

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:

- 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.

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, however, it is desirable to store a few non-homogeneous 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.

Overview of Sequences

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

Symbolics Common Lisp provides a range of general sequence functions that operate on both lists and 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 and 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:

- Constructing and accessing
- Predicates
- Mapping
- Modifying
- 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".

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. Nonprinting 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.

Using Modifier Keys

When any of the modifier bits (control, meta, super, or hyper) is set in conjunction with a letter, the letter is always uppercased.

The Control-Shift- characters are encoded separately. c-sh-A is not a synonym for c-A; they are distinct compound keystrokes. c-A names a gesture meaning to hold down the CONTROL key which pressing the A key.

In addition to the four modifier keys HYPER, SUPER, CONTROL, and META, the SHIFT key is a modifier key for letters when used in combination with one of the other modifiers. The CAPS LOCK key is not a modifier key and is always ignored in compound keystrokes. Thus typing CONTROL and A at the same time gives c-A; pressing CONTROL and SHIFT and / at the same time gives c-?, not c-sh-/.

The names for compound keystrokes always show a letter as capitalized. This does not mean that you have to use the SHIFT key; use the SHIFT key as a modifier only when sh- appears in the same name.

In addition, printing names of characters have case in them. Case is ignored on input. Some new synonyms for existing characters are accepted. In particular, names of the following form have these synonyms:

Name	*Equivalent to*
#\c-sh-B	#\c-shift-B
#\mouse-L	#\mouse-L-1

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 LGP3 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, and 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".

Overview of Strings

The Lisp data type, *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".

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

Zetalisp Note: 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".

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 strings 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). 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 MICE")  => "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 colon (:). The most important are keyword arguments **: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
```

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".

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.

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* &optional *variable-name* — Causes the cell at *loc* to become unbound.

locativep *x* — Tests if *x* is a locative.

locf *reference* — Converts *reference* to a new form that creates a locative pointer to that cell.

Overview of Table Management

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 that 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*. More complex tables can have 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 a bird of that type 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.

Genera's 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, your code is easily ported to Common Lisp, and you take advantage of the efficiencies provided by the facility. If you create tables that do not use the Symbolics extensions to the **make-hash-table** function, your code is already compatible with Common Lisp.

Note: In figuring out the best internal representation for the given data, the table management facility uses a small amount of overhead. Thus, if you know beforehand that you need a simple table, for instance a property list or an association list, it may be more efficient to create your own list rather than use the table management facility to do it.

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))
   => #<Table 0/0 63151256>

(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)
   => #<Table 2/2 63151256> 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
   #< #<Table 2/2 63151256>
```

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. It defines a large family of table flavors, with generic functions for accessing them. This makes it easy to use, as well as flexible and extensible.

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".

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". The encapsulation is composed of function-

altering functions, such as **trace** and **advise**. See the section "Encapsulations". 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".

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".

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".

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".

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, **zl:closure**. For more information on dynamic closures: See the section "Dynamic Closures".

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". A full description of each predicate is available in the dictionary of Lisp functions.

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.

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". 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".

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".

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.

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 symbol has no functional value, an error is signaled. 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
```

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".) 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.

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 is:

```
(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)))))
```

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 that 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. All of the keywords for **loop** are Symbolics Common Lisp extensions to the language specification in Guy L. Steele's *Common Lisp: the Language*.

Here are some examples showing how loop keywords can be used:

```
(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. Using the appropriate keywords helps you 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.

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 the 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".

See the section "Overview of Flavors".

See the section "Comparing **defstruct** Structures and Flavors".

Overview of CLOS

Introduction to CLOS

The Common Lisp Object System (CLOS) enables users to program in an object-oriented style within Common Lisp. CLOS is part of the draft ANSI specification of Common Lisp.

Symbolics continues to support New Flavors, another object-oriented language. The primary advantage of CLOS over Flavors is that CLOS is a standard part of ANSI Common Lisp, and thus CLOS programs can be ported to other platforms. CLOS offers some extra functionality which users will find valuable, and omits some of the less vital functionality of Flavors. Users can continue to develop programs in Flavors if they are not interested in developing portable code, do not need the extra features that CLOS offers, or have programs that need to access flavors.

We do not support programming in a style that mixes use of CLOS and Flavors. That is, CLOS classes cannot inherit from flavors (and vice versa), and you cannot call a CLOS generic function on a Flavors instance (and vice versa).

Classes, Types, and Instances

You can define new classes to represent objects that your program is modeling. Each individual object is represented by an instance of the class. Each class has a type associated with it.

In CLOS, every Lisp object is an instance of a class. You can use **clos:class-of** to determine the class of any Lisp object.

In addition to user-defined classes, CLOS has a set of predefined classes. CLOS defines classes that correspond to many Lisp types, including classes named **array**, **integer**, **list**, **t**, and others. (Note that not all types have associated classes.) Since methods can specialize on these predefined classes, CLOS enables the object-oriented programming style to encompass many useful Lisp types as well as user-defined types.

Slots

All instances of a class have the same structure, which is represented by its *slots*. A slot has a name and a value. Slots are used to store state information about an object.

CLOS enables you to read and write the value of a slot using *accessors*. A *reader* is an accessor function that returns the value of a slot. A *writer* is an accessor function that sets the value of a slot.

CLOS supports two kinds of slots:

Local slot — Each instance of the class stores its own value for a local slot. In other words, the storage for the slot is allocated on a per-instance basis. Local slots are used for state information which should be associated with each individual instance.

Shared slot All instances of the class share the value of a shared slot. The storage for the slot is allocated on a per-class basis. Shared slots are used for state information which should be associated with all instances of the class.

Class Inheritance

CLOS enables you to define classes that *inherit* from other classes. Inheritance is a key aspect of the object-oriented paradigm; it enables you to conveniently model similar kinds of objects that have minor differences. You can identify shared aspects of these objects, and isolate each aspect in a discrete module, which you might consider a "building block" class. You can then combine these modules to create new classes. The sharable aspects are defined and implemented once, and are included in the classes that should exhibit those behaviors.

CLOS supports *multiple inheritance*, which means that a class can inherit from any number of "parent" classes. In contrast, note that **defstruct** supports only single inheritance; there can only be one parent structure included in the definition of a new type defined by **defstruct**.

When you define a class, you specify its *direct superclasses*. The new class inherits from all its direct superclasses, and from all their direct superclasses, and so on. The set of classes that the class inherits from is called its *superclasses*. The complementary terms are *direct subclasses* and *subclasses*.

Suppose:

`Class-A` inherits from `Class-B` and `Class-C`.
`Class-C` inherits from `Class-D`.
`Class-B` inherits from `Class-E`.

Then:

`Class-A` is a direct subclass of `Class-B` and `Class-C`.
`Class-A` is a subclass of `Class-B`, `Class-C`, `Class-D`, and `Class-E`.

`Class-B` and `Class-C` are direct superclasses of `Class-A`.
`Class-B`, `Class-C`, `Class-D`, and `Class-E` are superclasses of `Class-A`.

A class inherits slots and other characteristics from its superclasses.

CLOS computes a *class precedence list* for each class. The purpose of the class precedence list is to ensure an orderly and predictable inheritance behavior, especially in cases of potential conflict, where more than one class specifies a certain characteristic.

The class precedence list is a list of the class itself and all its superclasses, in a precedence order from *most specific* to *least specific*. Each class has precedence over the classes that follow it in the class precedence list. In other words, each class is more specific than the classes that follow it in the class precedence list. The class precedence list for **Class-A** is:

```
(Class-A Class-B Class-E Class-C Class-D clos:standard-object t)
```

Notice the classes **clos:standard-object** and **t**, which appear at the end of the class precedence list. The predefined class **clos:standard-object** is automatically included as a superclass of each user-defined class; it supports the default behavior of user-defined classes. The predefined class **t** is automatically a superclass of every class (both user-defined classes and predefined classes); it appears as the last class in every class precedence list.

Generic Functions

A *generic function* is called with the same syntax as an ordinary Lisp function. The difference lies in what happens when the function is called. An ordinary function has a single body of code that is always executed when the function is called. When a generic function is called, the body of code that is executed depends on the arguments to the generic function.

A generic function can have several *methods*, each with its own body of code; the arguments to the generic function cause one or more of the methods to be invoked. The combined body of code (which consists of one or more methods) is the *effective method*, sometimes called the "handler". When a generic function is called, the CLOS *generic dispatch* mechanism automatically chooses and executes the appropriate effective method for each generic function call.

The CLOS model focuses on generic functions, which is an important difference from other object-oriented systems, which focus on a class and methods for that class. In many object-oriented systems (including Flavors), the effective method of a generic function is chosen based on a single argument to the generic function, which means that each method is associated with a single class. Another way to think about this is that Flavors is "class-centric" and CLOS is "generic-function-centric".

In CLOS, any one or more of the required arguments to the generic function can select methods to be combined into the effective method. Each method can be associated with a number of classes, up to the number of required arguments.

Methods

Methods perform the work of generic functions. The important concepts of CLOS methods are:

- The method's *applicability*. The lambda-list of the method states the sets of arguments for which the method is applicable. Each required parameter can be *specialized*. Each specialized parameter is an applicability test; a method is applicable if all the specialized parameters are satisfied by the arguments to the generic function.

  A parameter can be specialized in two ways:

- ° With a class name. To satisfy this applicability test, the argument must be an instance of the class or an instance of any subclass.

- ° With a list such as (**eql** *form*). To satisfy this applicability test, the argument must be **eql** to the Lisp object that is the value of *form*. Note that the *form* is evaluated once, at the time that the **clos:defmethod** form is evaluated. The *form* is not evaluated each time the generic function is called.

Note that generic functions can accept optional, rest, and keyword arguments as well as required arguments, but only required arguments participate in method applicability.

- The method's *role*. Each method has a *qualifier* that states the role that the method plays in the generic function, and how it fits in with other methods. We discuss some of the common method roles:

  The role of a *primary method* is to perform the main work of the generic function.

  There might be other methods that perform additional or auxiliary work; these include *before-methods* and *after-methods*. Before-methods run before the primary method, to do preparatory or initialization work. After-methods run after the primary method, to do clean-up work.

  An *around-method* has special control; it can decide whether the primary method should be executed; it can provide code that runs before the before-methods and code that runs after the after-methods; and it can bind state around the call of the other methods.

  In addition to these roles, users can define new roles customized for a particular application. For more information: See the section "CLOS Method Combination".

  CLOS supports both the *declarative style* of programming (where before-methods, primary methods, and after-methods are used, and each method is called automatically when appropriate) and the *imperative style* (where the body of an around-method uses **clos:call-next-method** to call a method imperatively). For a discussion of these two styles, see "Controlling the Generic Dispatch" in the book *Object-Oriented Programming in COMMON LISP*.

- The method's *body*. The method's body consists of Lisp forms that perform some work of the generic function.

Method-Combination Types

Each generic function has a *method-combination type*, which controls the interaction between different kinds of methods. The method-combination type controls:

- Which method roles are supported.

- How the methods of the different roles work together in the generic function.
- How the value or values of the generic function are calculated.

The default method-combination type is called **clos:standard**. It supports primary methods, before-methods, after-methods, and around-methods, as described above.

CLOS provides a set of predefined method-combination types (in addition to **clos:standard**), and it also provides a mechanism for users to define new method-combination types.

Generic Dispatch

The *generic dispatch* ties together all of the concepts mentioned above; it controls the behavior of generic functions. The generic dispatch is an automatic mechanism of CLOS that selects and executes the appropriate effective method of a generic function based on the arguments to the generic function.

When a generic function is called, the CLOS generic dispatch does the following:

1. Finds the set of applicable methods. A method is applicable if each of its required parameters is satisfied by the corresponding argument to the generic function.

2. Arranges the applicable methods in precedence order. The precedence order of methods is calculated based on the parameter specializers of the methods, and the class precedence lists of the required arguments to the generic function.

3. Uses the method-combination type of the generic function to determine how the applicable methods should be combined into an effective method. (The effective method is the body of code that CLOS constructs to perform the generic function for the given arguments.) The method-combination type uses the sorted list of applicable methods as its input. It decides which methods should be executed, and in what order.

4. Executes the effective method and returns its values.

Note that the Symbolics CLOS implementation optimizes the generic dispatch, so that some of the steps of the generic dispatch are not executed on each generic function call. The optimizations, however, do not change the semantic effect of the generic dispatch procedure as described above.

CLOS Objects and Meta-Objects

The basic elements of CLOS programs are implemented by first-class objects; for example, there are *class objects*, *generic function objects*, and *method objects*. These objects are distinct from their names. Most operators in the CLOS Programmer Interface enable you to deal with objects by using the names of the objects; for example, you can refer to generic functions and classes by their names.

Underlying the CLOS Programmer Interface is another level called the Meta-object Protocol. The Meta-object Protocol is not currently part of the draft ANSI standard for Common Lisp. Symbolics CLOS does not implement the complete Meta-object Protocol, but it does support some of its features. The purpose of the Meta-object Protocol is to define CLOS itself in an extensible, object-oriented way, such that users can develop different object-oriented paradigms, or paradigms that modify or extend CLOS.

In the Meta-object Protocol, class objects, generic function objects, and method objects (among others) are implemented as instances of classes. Instances of these classes are called *meta-objects*. Here are some of the predefined meta-objects:

- The default class of user-defined classes is **clos:standard-class**.
- The default class of generic functions is **clos:standard-generic-function.**
- The default class of methods is **clos:standard-method**.

A class whose instances are classes is called a *metaclass*. The class **clos:standard-class** is a metaclass, because its instances are user-defined classes. There are two other metaclasses of interest:

- The class of most of the predefined classes that correspond to Common Lisp types (such as **list**) is **clos:built-in-class**.
- The class of classes defined by **defstruct** is **clos:structure-class**.

Basic Use of CLOS

This section introduces the basic CLOS operators and shows a brief example of using them.

Basic CLOS Operators

When developing an object-oriented program, the most common things you will need to do include defining classes, defining generic functions, defining methods, and creating new instances.

clos:defclass *class-name superclass-names slot-specifiers* &rest *class-options*
Defines a class named *class-name*, and returns the class object.

clos:defgeneric *function-name lambda-list* &body *options-and-methods*
Defines a generic function and returns the generic function object.

clos:defmethod *function-name {method-qualifier}* specialized-lambda-list* &body *body*
Defines a method for a generic function and returns the method object.

clos:make-instance *class* &allow-other-keys
Creates, initializes, and returns a new instance of the given class.

Simple Example of Using CLOS

This section presents a simple example of using the basic CLOS operators to define classes, make instances, call accessors, define generic functions and methods. Each of these subjects is covered in detail elsewhere in the documentation; this section is intended only to give you an idea of what a CLOS program looks like.

Defining a Class

We might define a new class called **person** as follows:

```
(defclass person ()
  ((name :initarg :name :accessor name-of)
   (ssn :initarg :ssn :accessor soc-sec-number
        :documentation "social security number")
   (address :initarg :address :accessor address)))
```

The class **person** has no superclasses (the second subform of **clos:defclass** is an empty list). It has three local slots, named **name**, **ssn**, and **address**. Each slot has some slot options. The slot options used here have the following effect:

:initarg Enables us to initialize this slot when making an instance. Here, the initialization arguments are **:name**, **:ssn**, and **:address**.

:accessor Defines two methods: a method for a reader and a method for a writer generic function, which we can use to access the value of the slot. Here, the readers are named **name-of**, **soc-sec-number**, and **address**. We can write the value of one of these slots by using **setf** with the reader.

:documentation Documents the slot.

For more information on defining classes: See the macro **clos:defclass**. See the section "CLOS Classes and Instances".

Making an Instance

We can make an instance of **person** as follows:

```
(setq *constance*
      (make-instance 'person :name "Constance McGill"
                             :ssn "012-34-5678"))
```

Notice that we initialized the value of the **name** slot by providing the **:name** initialization argument to **clos:make-instance**. Similarly, we initialized the value of the **ssn** slot by providing the **:ssn** initialization argument.

We did not initialize the value of **address**, so that slot's value is unbound.

For more information on making instances: See the section "Creating and Initializing CLOS Instances".

Calling Accessor Generic Functions

We can read the value of the **name** and **ssn** slots of the instance by calling the readers as follows:

```
(name-of *constance*) => "Constance McGill"
(soc-sec-number *constance*) => "012-34-5678"
```

The writers are **setf** generic functions that must be called with the **setf** syntax. Here, we write the value of the **address** slot:

```
(setf (address *constance*) "44 Pine St")
```

For more information on accessing slots: See the section "Accessing Slots of CLOS Instances".

Defining Classes that Inherit from Other Classes

We can define the class **employee** in such a way that it inherits from the class **person**:

```
(defclass employee (person)
  ((salary :initarg salary :accessor salary)
   (vacation-time :initform 0 :accessor vacation-time)
   (phone :reader phone-extension)
   (rank :initarg rank :accessor rank)))
```

The class **person** is a direct superclass of **employee**. Conversely, the class **employee** is a direct subclass of **person**.

The class **employee** inherits three local slots from **person**, and specifies four additional slots of its own.

We see two new slot options in this definition:

:initform Gives a default initial value for the slot.

:reader Defines a reader method, but no writer method.

The slot **vacation-time** has no initialization argument, so we cannot initialize it by giving an argument in the call to **clos:make-instance**. Instead, this slot is always initialized to the value of its initform, which is **0**.

We might need a class to represent employees who are in the Human Resources staff. We can define the class **H-R-staff** as a subclass of **employee**:

```
(defclass H-R-staff (employee) ()
  (:documentation "H-R-staff have authority to alter records."))
```

This class inherits four slots from **employee**, and three slots from **person**, but adds no other slots. It uses the **:documentation** class option to document the class as a whole.

For more information on how class inheritance works: See the section "CLOS Inheritance".

Defining a Generic Function and Methods

Here we define a generic function called **change-name**:

```
(defgeneric change-name (employee staff new-name)
  (:documentation "Ensures that name change is done by authorized staff."))
```

At this point, the generic function is defined, but there are no methods defined for it. If it is called with any set of arguments, an error will be signaled, stating that there are no applicable methods. Thus, the next step is to define methods for this generic function.

H-R staff people are authorized to change an employee's name. The following method for **change-name** is applicable when the first argument is of the type **employee** and the second argument is of the type **H-R-staff**. The body of the method changes the value of the employee's name to a new name.

```
;;; Method intended to be called when an H-R person
;;; tries to change an employee's name.
(defmethod change-name ((emp employee) (h-r H-R-staff) new-name)
  (setf (name-of emp) new-name))
```

The following method for **change-name** is applicable when the first argument is of the type **employee**. The second and third arguments set no restrictions on the applicability of the method. The intention is for this method to be called when the second argument is a person who is *not* authorized to change an employee's name; it signals an error instead of changing the employee's name.

```
;;; Method intended to be called when a non-h-r person
;;; tries to change an employee's name.
(defmethod change-name ((emp employee) non-h-r new-name)
  (declare (ignore non-h-r new-name))
  (error "You aren't authorized to change an employee's name."))
```

If **change-name** is called with an employee as its first argument and a H-R-staff person as its second argument, then both methods are applicable. The first method is more specific than the second. Thus, only the first method is called, and it changes the name of the employee.

If **change-name** is called with an employee as its first argument and a non-H-R-staff person as its second argument, only the second method is applicable. That method signals an error, because it is enforcing the principle that only authorized staff can change an employee's name.

At present, our model is that only H-R staff people are authorized to change an employee's name. Thus we have two methods: one intended to be called for H-R staff people, and the other for other employees. Later on, we might decide that people in the accounting department are also authorized to change an employee's name. We could define a primary method applicable for people in the accounting department which would do the same thing that the method for H-R-staff does.

Note that we assume that users do not call **setf** of **name-of** directly, because it is not part of the advertised interface; calling it directly would bypass this error-checking. CLOS does not include any protection features that would guard against users calling **setf** of **name-of** directly.

For more information on methods and generic functions:

See the section "CLOS Methods and Generic Functions".

See the section "CLOS Method Combination".

See the macro **clos:defmethod**.

See the macro **clos:defgeneric**.

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.

If you are developing code with the intention of porting it to other Lisps, you should use CLOS instead, The primary advantage of CLOS over Flavors is that CLOS is a standard part of ANSI Common Lisp, and thus CLOS programs can be ported to other platforms. CLOS offers some extra functionality which users will find valuable, and omits some of the less vital functionality of Flavors. Users can continue to develop programs in Flavors if they are not interested in developing portable code, do not need the extra features that CLOS offers, or have programs that need to access flavors.

We do not support programming in a style that mixes use of CLOS and Flavors. That is, CLOS classes cannot inherit from flavors (and vice versa), and you cannot call a CLOS generic function on a Flavors instance (and vice versa).

For an introduction to CLOS, see the section "Overview of CLOS". For reference information on CLOS, see the section "Symbolics CLOS".

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".

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.

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 method 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". See the section "Using Message-Passing Instead of Generic Functions".

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.

Generic functions are the 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 has 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".

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".

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
```

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 an important strength 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 Programs — *Single Protocol* — *Various 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.

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 reference a slot of a **defstruct** structure.	It is somewhat slower to 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**.

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 neces-

	sarily 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.

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".) 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, if both your program and the compiler have a function called **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".

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". 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**.

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".

Overview of the I/O System

Symbolics Common Lisp provides a powerful and flexible system for performing input and output to peripheral devices. To allow device-independent I/O (that is, to allow programs to be written in a general way so that the program's input and output may be connected with any device), the I/O system provides the concept of an "I/O stream". What streams are, the way they work, and the functions to create and manipulate streams, are described in this document. This document also describes the Lisp "I/O" operations **zl:read** and **print**.

Data Types

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.

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 ! 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.

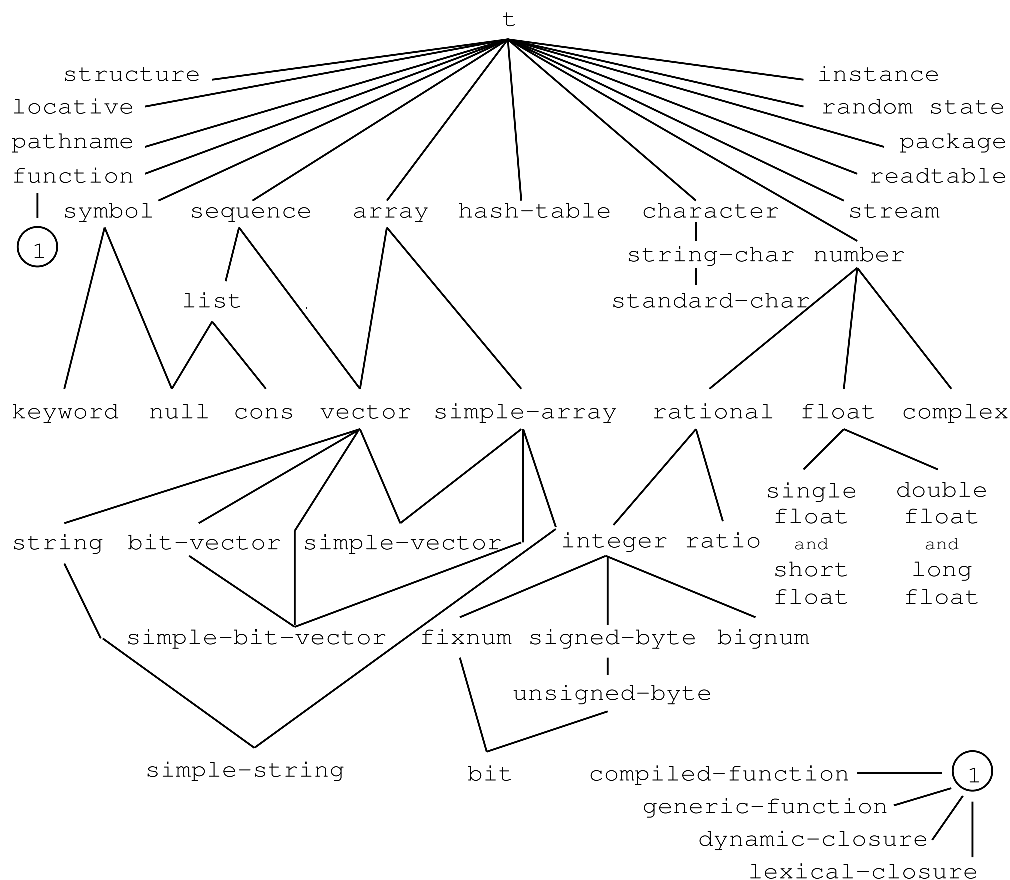


Figure 3. Symbolics Common Lisp Data Types

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".

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.

Some Major Data Types

Here are brief descriptions of the top level and a few lower-level Symbolics Common Lisp data types. Most of 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 the 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 them 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 specialized to hold only characters are called strings. One-dimensional arrays specialized to hold only 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 section "Naming of Files".

- *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 section "Streams".

- *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 new flavor 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* are the usual form of compiled, executable Lisp code. A compiled function contains the code for one function. Compiled functions are pro-

duced 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 **zl:closure** function and the **zl: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, are never used by most programmers.

These data types are not always mutually exclusive.

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".

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

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 document *Symbolics Common Lisp Dictionary*.

array	**instance**	**short-float**
atom	**integer**	**simple-array**
bignum	**keyword**	**simple-bit-vector**
bit	**list**	**simple-string**
bit-vector	**sys: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**
sys:dynamic-closure	**pathname**	**structure**
fixnum	**random-state**	**symbol**
float	**ratio**	**t**
function	**rational**	**unsigned-byte**
sys:generic-function	**readtable**	**vector**
hash-table	**sequence**	

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.
- Constructing abbreviated forms of type specifiers.

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 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, you would write:

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

To leave the element type unspecified, you 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`.

Predicating Type Specifiers

A type specifier list of the following form lets you define the set of all objects that satisfy the predicate named by *predicate-name*:

```
(satisfies 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 following type is the same as the type **number**:

```
(satisfies numberp)
```

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)))
```

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 appropriate type specifier lists for this purpose:

(member &rest *list***)** Denotes the objects that are **eql** to one of the specified objects in *list*.

(not *type***)** Denotes 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.

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*:

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

This list format 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 specialized arrays of the type *element-type*, and whose dimensions match *dimensions*, a list of integers.

(sequence *type***)** Denotes the sequences 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 specialized one-dimensional arrays 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 *argn-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.

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 non-negative integers less than *n*.

(ratio *low high***)** Denotes the ratios between *low* and *high*. This is a Symbolics Common Lisp extension to Common Lisp.

(signed-byte *s***)** Denotes the 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 floating-point numbers between *low* and *high* exclusive.

(string *size***)** Denotes the strings of the indicated *size*.

(simple-string *size***)** Denotes the 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 simple-bit-vectors of the indicated *size*.

(sequence *type***)** Denotes the sequences of the type *type*. *type* is a Symbolics extension to Common Lisp.

(sequence *type***)**

is the same as:

```
(or vector list)
```

The following examples, which are equivalent, create a vector that contains only **string-char**s:

```
(sequence string-char)
```

and:

```
(or (declare (vector string-char))
    list)
```

The following are subtypes of **(sequence string-char)**:

```
list
(vector string-char)
(vector character)
(vector t)
```

Type Modifiers for Vector and Array Types

The type specifier **declare** indicates a type specifier list used for identifying a specific kind of array. For example:

```
(setq array (make-array 5 :element-type '(integer 0 99)))
(array-element-type array) => (unsigned-byte 8)
```

(unsigned-byte 8) is the smallest size for an array that can contain integers from 0 to 99. **(integer 0 99)** is a subtype of **(unsigned-byte 8)**. For example:

```
(typep array
       '(vector (integer 0 99))) =>nil
```

The **vector** type, as specified in Common Lisp, indicates an array specialized to contain only elements of the type **(integer 0 99)**. This is only sometimes what you want. In this example, array is not specialized to hold only **(integer 0 99)**, it is specialized to hold **(unsigned-byte 8)**:

```
(typep array
       '(declare (vector (integer 0 99)))) => t
```

In this example, array is capable of containing **(vector (integer 0 99))**.

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 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.

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**.

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 in, among other things, controlling program flow and error-checking.

There are also numerous specialized predicates for type-checking. See the section "Predicates". 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 for 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 an object of a type specified by a Common Lisp.
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".

ctypecase *object* &body *body* — "Continuable exhaustive type case." Like **typecase**, but signals a proceedable error if no clause is satisfied. See the section "Conditionals".

etypecase *object* &body *body* — "Exhaustive case." Like **typecase**, but signals a non-proceedable error if no clause is satisfied. See the section "Conditionals".

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".

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**

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".

Type Specifiers in the CL Package with SCL Extensions

Here are the type specifiers that have Symbolics Common Lisp extensions:

Type specifier	*Extension(s)*
number	*low-limit*, *high-limit*

ratio *low*, *high*

sequence *type*

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.

Types of Numbers

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

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" as 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 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".

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".

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 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 types have the same printed representations.

A few "low-level" functions work only on fixnums, and some built-in system functions 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. In Symbolics Common Lisp the range is from -2147483648 to 2147483647 (-2^{31} to $2^{31}-1$).

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

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 is appropriate for situations where there is no exact rational answer to a problem (for instance **pi** or, **(sqrt 2)**), or where exact answers are not required. When using floating point, the approximate nature of the representation must be kept in mind. See the section "Non-mathematical Behavior of Floating-point Numbers".

The internal representation of floating-point numbers uses a mathematical sign $s \in \{+1,-1\}$, a significand (fraction part) 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. These are the IEEE standard single- and double-precision formats. See the section "IEEE Floating-point Representation".

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.

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.

See the section "Numeric Type Conversions".

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.

IEEE Floating-point Representation

Genera 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.

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.

Constants Indicating the Range of Floating-point Numbers

Constant	*Value*
least-positive-single-float	1.4e-45
least-positive-normalized-single-float	1.1754944e-38
most-positive-single-float	3.4028235e38
least-negative-single-float	-1.4e-45
least-negative-normalized-single-float	-1.1754944e-38
most-negative-single-float	-3.4028235e38
least-positive-double-float	5.0d-324
least-positive-normalized-double-float	2.2250738585072014d-308
most-positive-double-float	1.7976931348623157d308
least-negative-double-float	-5.0d-324
least-negative-normalized-double-float	-2.2250738585072014d-308
most-negative-double-float	-1.7976931348623157d308

Since the exponent in floating-point representation has a fixed length, some numbers cannot be represented. Thus 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 standard result for the mathematical result.

Example:

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

Non-mathematical Behavior of Floating-point Numbers

The restricted representation of floating-point numbers leads to much behavior which can be confusing to users unfamiliar with the concept. This behavior is characteristic of floating-point numbers in general, and not of any particular language, machine, or implementation.

Floating-point operations don't always follow normal mathematical laws. For example, floating-point addition is not associative:

```
(+ (+ 1.0e10 -1.0e10) 1.0) => 1.0
(+ 1.0e10 (+ -1.0e10 1.0)) => 0.0
```

This follows from the restricted representation of floating-point, since 1.0 is insignificant relative to 1.0e10.

Much of the confusion surrounding floating-point comes from the problem of converting from decimal to binary and vice versa.

Consider that the binary representation of 1/10 repeats infinitely:

```
.0001100110011001100110011001100110011001100110011001100 ...
```

Since we can't represent this exact value of 1/10, we would like to find the mathematically closest number which is representable. We do that by rounding to the appropriate number of binary places:

```
Single precision:  (24 significant bits)
.000110011001100110011001101

(describe (float 1/10 0.0)) =>
0.1 is a single-precision floating-point number.
 Sign 0, exponent 173, 23-bit fraction 23146315  (not including hidden bit)
 Its exact decimal value is 0.100000001490116119384765625
0.1

Double precision:  (53 significant bits)
.00011001100110011001100110011001100110011001100110011010
```

```
(describe (float 1/10 0.0d0)) =>
0.1d0 is a double-precision floating-point number.
 Sign 0, exponent 1773, 52-bit fraction 114631463146314632  (not including hidden bit)
 Its exact decimal value is 0.1000000000000000055511151231257827021181583404541015625d0
0.1d0
```

Already we see some anomalies. The single-precision number closest to 1/10 has a different mathematical value from the double-precision one. So a decimal number, when represented in different floating-point precisions, can have different values. Yet the printer prints both as "0.1".

Why do the printed representations hide the difference in values? Every binary number has an exact, finite, decimal representation, which can be printed. The **describe** function does that, as shown in the example above. From that example, you can see that printing exact values would be cumbersome without giving useful information. So the printer prints the shortest decimal number that is properly rounded (from the actual decimal value), and whose rounded binary value (in that precision) is identical to the original.

Here is an example of the rule used to derive the shortest decimal number:

```
(describe 1.17) =>
1.17 is a single-precision floating-point number.
Sign 0, exponent 177, 23-bit fraction 05341217  (not including hidden bit)
Its exact decimal value is 1.16999995708465576171875
1.17
```

The correctly rounded decimal values for this single-precision number are:

```
1, 1.2, 1.17, 1.16999996, 1.169999957, 1.1699999571, 1.16999995708, etc.
```

Rounded to single-precision (binary), the first three printed representations are all different, but after 1.17, they are all the same. Thus, 1.17 is the "best" representation to print.

Since the printing rule is sensitive to floating-point precision, it hides the difference between the exact mathematical values of 1.17 and 1.17d0.

The interactions between the printing rule and the finite representation of numbers (both as read in and as computed) can lead to some interesting anomalies:

```
(- 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
```

These are all "correct", as we can verify by doing the exact (rational) arithmetic.

```
(rational 6) => 6
(rational 5.9) => 12373197/2097152
(- 6 12373197/2097152) => 209715/2097152
(float 209715/2097152 0.0) => 0.099999905
```

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".

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)
```

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

Type Specifiers and Type Hierarchy for Numbers

The type specifiers relating to numeric data types are:

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 ! 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".

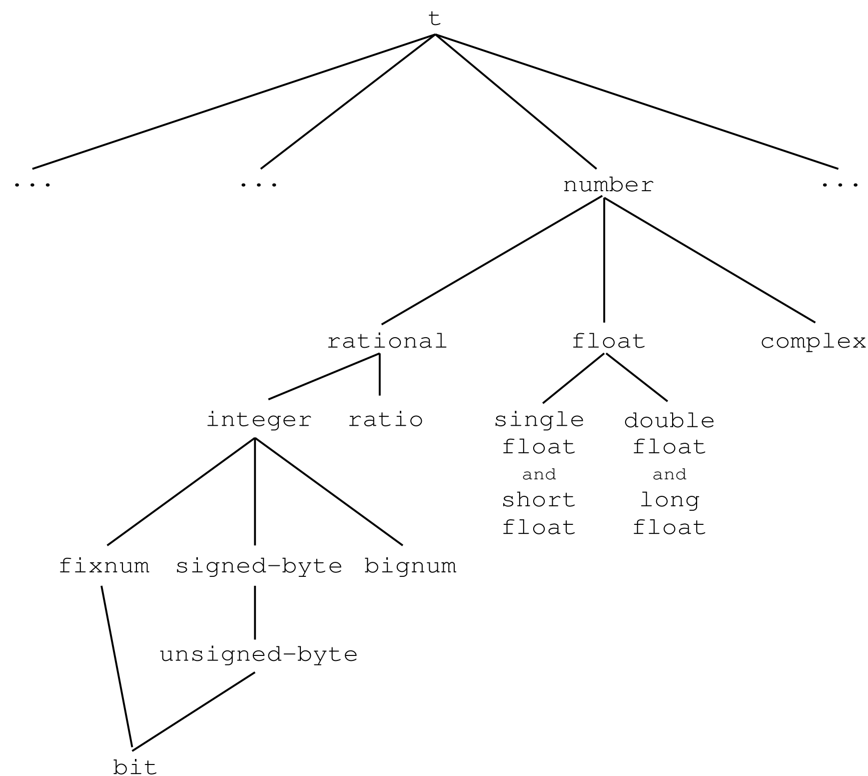


Figure 4. Symbolics Common Lisp Numeric Data Types

How the Printer Prints Numbers

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

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

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".

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 in the following sequence:

- 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.

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
```

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 in the following sequence:

- 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 that corresponds to the current value of the 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**, or **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 should always produce an equal floating-point number.

The printed representation for floating-point "infinity" is in the following sequence:

- 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∞
```

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.

Note: The following Zetalisp variable is included to help you read old programs. In your new programs, use the Common Lisp version of *this variable.*

zl:base The value of **zl:base** is a number that is the radix in which integers and ratios are printed in, or a symbol with a **si:princ-function** property. The Common Lisp equivalent of this variable is ***print-base***.

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)
```

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 values of control variables, as explained below.

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. The reader accepts radix specifier syntax in both upper and lowercase characters. See the section "Radix Specifier Format".

- 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**.)

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
```

How the Reader Recognizes Integers

The syntax for a simple integer is the following sequence:

- 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
```

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
```

Compatibility Note: The fact that the reader depends on the value of these variables to tell it how to interpret tokens when the value of ***read-base*** is greater than ten, rather than just automatically interpreting them as numbers, is an incompatible difference from the language specification in *Common Lisp: The Language*.

How the Reader Recognizes Ratios

The syntax of a ratio is the following sequence:

- 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".

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
```

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
- Some optional 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
- An optional 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.

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 how different 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 the following sequence:

- A required plus or minus sign
- The digit "1"
- Any of the exponent mark characters
- 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∞
```

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).

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.

Note: The following Zetalisp variable is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this variable.

Zetalisp	*Common Lisp*
zl:ibase	The value of **zl:ibase** is a number that is the radix in which integers and ratios are read. The Common Lisp equivalent of this variable is ***read-base***.

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)
```

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 that check numeric type and properties.
- Functions which perform numeric comparison.
- Arithmetic functions allowing numeric data conversions.
- Transcendental functions and a pseudo-random number generator facility.

- Functions that 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.

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 coer-

cions 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)
```

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 Dictionary*.

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.

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".

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.

sys:fixnump *object* Tests for fixnum.

Note: The following Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:bigp *object* Tests for bignum.

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.

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 number is zero.

Note: The following Zetalisp predicate is included to help you read old programs. In your new programs, if possible, use the Common Lisp equivalent of this predicate.

zl:signp *test number* Tests if the sign of *number* matches *test*.

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.

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".
- 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** and **equal** and 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)))))
```

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

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".

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 integers 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".

Arithmetic Functions

Function	*Action*
+ &rest *numbers*	Returns the sum of its arguments.
- *number* &rest *more-numbers*	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*	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*	Adds 1 to *number*.
1- *number*	Subtracts 1 from *number*
gcd &rest *integers*	Returns the greatest common divisor of all its arguments.
lcm &rest *integers*	Returns the least common multiple of all its arguments.
print-exact-float-value	When this variable is set to **t**, it prints the exact number represented by a floating-point number, not the rounded number, which is normally printed by the printer.
rem *number divisor*	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.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:+$
zl:plus — Synonyms for **+**.

zl:-$ — Synonym for **-**.

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:*$
zl:times — Synonyms for *.

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:1+$
zl:add1 — Synonyms for **1+**.

zl:1-$
zl:sub1 — Synonyms for **1-**.

zl:gcd *integer1 integer2* &rest *more-integers*
zl: — Returns greatest common divisor of all its arguments.

zl:
zl:remainder — Synonyms for **rem**.

zl:expt *num expt*
zl:^
zl:^$ — Returns *num* raised to the *expt* power.

zl:sqrt *n* — Returns the square root of *n*.

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: Given an integer, they convert it to a single-float and return a single-float; Given a double-float, they return a double-float.

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*.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp version of this function.

zl:log *x* — Returns the natural logarithm of *x*.

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.)

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:atan *y x* Returns angle between 0 and 2π in radians.

zl:atan2 *y x* Returns angle in radians (same as **atan**).

Hyperbolic Functions

sinh *number* Returns the hyperbolic sine of *number*.

cosh *number* Returns the hyperbolic cosine of *number*.

tanh *number* Returns the hyperbolic tangent of *number*.

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*.

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**".

In addition, there are functions that 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.

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.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this function.

zl:rational *x* — Converts a noncomplex number to an equivalent rational number. **rationalize** is the Common Lisp equivalent of this function.

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 returns *number*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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.

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*.

*See the section "Comparison of **floor**, **ceiling**, **truncate** and **round**". *Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.*

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**).

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 an integer.

fround *number* &optional (*divisor* **1**)
Like **round**, except result is a floating-point number instead of an integer.

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.

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 ($float * 2^{integer}$).

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*.

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. Examples:

```
(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

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 that perform a specified bit-wise logical operation on their arguments and return the result, those that return specific components or characteristics of their argument, and a group of predicates based on bit-testing.

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.

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 the 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*.

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

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:logand *number* &rest *more-numbers*
Returns the bit-wise logical *and* of its arguments.

zl:logior *number* &rest *more-numbers*
Returns the bit-wise logical *inclusive or* of its arguments.

zl:logxor *number* &rest *more-numbers*
Returns the bit-wise logical *exclusive or* of its arguments.

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*

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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**).

Predicates for Testing Bits in Integers

logbitp *index integer*
Returns t if *index* bit in *integer* (the bit whose weight is 2^{index}) is a one-bit.

logtest *integer1 integer2*

Returns **t** if any 1-bits in *integer1* are 1-bits in *integer2*.

Note: The following Zetalisp predicate is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this predicate.

zl:bit-test *x y* Returns **t** if any 1 bits in *x* are 1 bits in *y*. Use the Common Lisp function, **logtest**.

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

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.

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.

Random Number Generation

The functions in this section provide a pseudorandom number generator facility. The basic function 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 the 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
```

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.

Random Number Functions

make-random-state &optional *state*
: Returns a new object of type **random-state** from the uniform distribution over [0, number).

random-normal &optional *mean standard-deviation state*
: Returns a random number from the normal distribution with the specified *mean* and *standard-deviation*.

random *number* &optional *state*
: Returns a noncomplex number of the same kind as *number*.

random-state-p *object*
: Returns **t** if *object* is of type **random-state**.

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*.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalent of this function.

zl:random &optional *arg random-array* Returns a random integer.

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 that require this sort of arithmetic, such as hash functions and pseudo-random number generation.

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* Returns *number* shifted *count* bits, left or right, depending on the sign of *count*; bits shifted at either end are lost; requires fixnum arguments.

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.

Symbols, Keywords, and Variables

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".) 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". 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".

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.

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. See the section "How the Reader Recognizes Numbers". 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***.

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 in a *package*. See the section "Packages".

Symbols can start with digits; for example, **read** accepts one named "345T". If you want to put strange characters (such as lowercase letters, whitespace, parentheses, or reader macro characters) inside the name of a symbol, put a backslash before each strange character. If you want to have a symbol whose print-name looks like a number, put a backslash 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 backslashes, which must be quoted with backslash. Examples of symbols:

```
foo                  ab.cd
bar\(baz\)           ab\ cd
34w23                car54
\123                 123+
|XY-hsiang Kitty|    |and\|or|
```

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".)
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.

- Backslashes ("\") 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***.

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 *symbol*

Returns the print name of *symbol*.

string= *string1 string2* &key *(:start1* **0***) :end1 (:start2* **0***) :end2*

Checks to see if *string1* and *string2* are the same.

Note: The following Zetalisp functions are included to help you read old programs.

In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:get-pname *symbol*

Returns the print-name of *symbol*.

zl:samepnamep *x y*

Returns **t** if the printed representation of the two symbols *x* and *y* is the same.

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 "Kinds of Variables". 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 only on *symbols*. Thus they work on special variables but not on local or instance variables.

Functions for assigning a value to a symbol

set *symbol value*	Assign *value* to *symbol*

Functions for retrieving the value of a symbol

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

Functions for removing 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 value

boundp *sym*	Returns **t** if the special variable *sym* has a value.
variable-boundp *variable*	Returns **t** if *variable* has a value. Works on any kind of variable. Does not evaluate *variable*.

Functions for locating the value cell of a symbol

sys:variable-location *variable*	Return a locative pointer to the value cell of *variable*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:set-globally *var value*	Assign *value* to *var* as a global variable.
zl:symeval *sym*	Like **symbol-value**.
zl:symeval-globally *var*	Like **symbol-value-globally**.
zl:value-cell-location *sym*	Returns a locative pointer to *sym*'s internal value cell. Obsolete on local and instance variables.

The Function Cell of a Symbol

Every symbol 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 fol-

lowing functions are analogous to the similar value-cell-related functions. See the section "The Value Cell of a Symbol".

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*.
symbol-function *sym*	Returns the function definition of *sym*.
sys:function-cell-location *sym*	Returns a locative pointer to *sym*'s function cell.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:fset *sym definition*	Stores *definition* in the function cell of *sym*.
zl:fsymeval *sym*	Zetalisp equivalent of **symbol-function**.

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".

The Property List of a Symbol

Every symbol has an associated property list. See the section "Property Lists". 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.

Functions Relating to the Property List of a Symbol

symbol-plist *symbol*
: Returns the list that represents the property list of *symbol*.

getf *plist indicator* &optional *default*
: Searches for the property *indicator* on *plist*.

get-properties *plist indicator-list*
: Searches the property list stored in *plist* for any of the indicators in *indicator-list*.

remprop *symbol indicator*
Removes *indicator* from the property list in *symbol*.

remf *place indicator*
Removes *indicator* from the property list stored in *place*.

sys:property-cell-location *symbol*
Returns a locative pointer to *symbol*'s property list cell.

Note: The following Zetalisp functions are included to help you read old programs.

In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:plist *symbol* Returns the property list of *symbol*.

zl:putprop *sym value indicator*
Gives *sym* an *indicator*-property of *value*.

zl:setplist *symbol list*
Sets the property list of *symbol* to *list*.

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**.

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.

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 printname 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 *print-name* &optional *permanent-p*
: Creates an uninterned symbol with print-name *print-name*.

copy-symbol *symbol* &optional *copyprops*
: Creates an uninterned symbol with the same print-name as *symbol*.

gensym &optional *arg*
: Invents a print-name and creates a symbol with that print-name.

gentemp &optional *(prefix* **"t"***)* *package*
: Like **gensym** but also interns the new symbol.

sys:gensymbol &optional *(prefix* **"g"***)* *count*
: Invents a print-name using *prefix* and creates a symbol with that print-name.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:copysymbol *symbol* &optional *copyprops*
: Like **copy-symbol**.

zl:gensym &optional *arg*
: Like **gensym**.

Keywords

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". 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 can be used with a **case**, 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".

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 'sys: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. In some cases, Zetalisp uses keywords as type specifiers. However, Common Lisp does not use keywords as type specifiers. For example, when given an array object, **zl:typep** returns the keyword **:array**, whereas **type-of** returns **array**. The type specifiers corresponding to flavors and structures are not keywords. In Genera, keywords are used as Zetalisp type specifiers only for historical reasons. See the section "Type-checking Differences Between Symbolics Common Lisp and Zetalisp".

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.

Variables

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.

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".

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".

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".

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 **defconstant**, or by explicit declarations. See the section "Declarations". 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. You are not allowed 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 **defconstant** 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.

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:*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.

Standard variables are reset to their standard values after a warm boot.

These are the currently defined standard variables and 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**
prin1	**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 is that 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**. ***readtable*** is bound to **si:standard-readtable** whenever a break loop or debug loop is entered. ***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, **si:standard-readtable** would not be necessary. The top-level value of ***readtable*** could be used instead. However, if the machine is warm booted while ***readtable*** is bound, the top-level value of ***readtable*** is lost.

Examples:

- This example illustrates the use of binding ***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 **read**, ***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 ***readtable*** and **si:standard-readtable**, the value of **si:initial-readtable** should never be changed. In addition, the readtable that is the value of **si:initial-readtable** should never be modified, modifications should be made only to the readtable that is the value of **si:standard-readtable**. See the function **copy-readtable**.

See the section "The Readtable".

Special Forms for Setting Variables

setf *reference value* &rest *more-pairs*
Takes a form that *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing. When used with **aref**, stores a value into the specified array element.

psetf &rest *pairs*
Similar to **setf**, except that **psetf** performs all the assignments in *parallel*, that is, simultaneously, instead of from left to right.

setq &rest *vars-and-vals*
Sets *variable(s)* to *value(s)*.

psetq &rest *pairs*
Similar to **setq**, except that **psetq** performs all the assignments in *parallel*, that is, simultaneously, instead of from left to right.

multiple-value-setq *vars value*
For calling a function that is expected to return more than one value. *value* is evaluated, and the *vars* are *set* (not lambda-bound) to the values returned by *value*.

Note: The following Zetalisp special form is included to help you read old programs. In your new programs, if possible, use the Common Lisp equivalent of this special form.

zl:psetq &rest *rest*
Just like a **setq** form, except that the variables are set "in parallel"; first all the value forms are evaluated, and then the variables are set to the resulting values.

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 *body*
Like **let**, except it binds 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 *varlist* &body *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**.

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 **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.

multiple-value-bind *vars value* &body *body*

Note: The following Zetalisp special forms are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these special forms.

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.

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.

Note: The following Zetalisp special form is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this special form.

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.

Lists

Introduction to Lists

The basic concepts and terminology associated with lists are described elsewhere. See the section "Overview of Lists". In brief, lists and list-like structures exist to organize data in tabular structures. The basic data type upon which all tabular structures are based is a structure with two components, called a **cons**; the head (*car*) of the cons can hold any Lisp object, and the tail (*cdr*) of the cons points to another Lisp object.

In a list, the car of the cons points to an element in the list and the cdr of the cons points to a list containing the rest of the list. The cdr of the last cons of the list points to **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. A *true list*, then, is a chain of conses linked by their *cdr* components and terminated by **nil**. See figure ! for an illustration of the list (a b c d e).

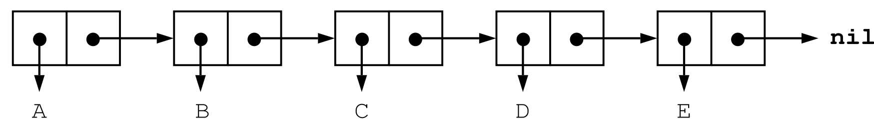


Figure 5. A List With Multiple Elements

A list is built recursively by adding a new element to the front of an existing list. This is done by creating a new cons whose car holds the element being added, and whose cdr points to the first element of the original list. For example, if you add a new cons whose car is the symbol f to the list (a b c d e), the new list (f a b c d e) is built. See figure ! for an illustration of the list (f a b c d e).

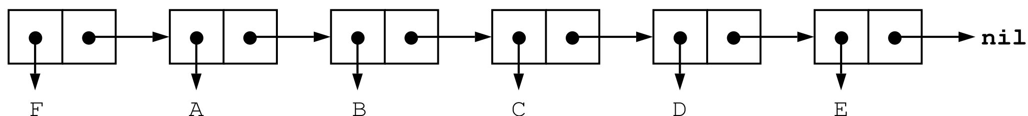


Figure 6. List With An Added Element

The symbol **nil** is used to represent the empty list, which is a list without any elements. The symbol **nil** and the list **()** are equivalent.

This chapter surveys the data types associated with lists, then discusses a variety of lists that can be built out of conses: simple lists, property lists, dotted lists, association lists, and trees, and their specialized operations. There is a brief discussion of the way the printer and the reader deal with lists; lastly we present the concept of cdr-coding, a special internal representation of conses for storage reduction.

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 a list with the symbol **nil** as its only object:

```
(nil)
```

For more information about conses: See the section "Overview of Lists".

list A sequence of linked conses, 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 (described below).

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 concepts 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 conses, where each cons is an association. The *car* of the cons is called the *key* (or *indicator*), and the *cdr* is called the *datum* (or *value*). For more information about association lists, see the section "Association Lists".

car
: This is the first element of a cons. It can be any Lisp object, for example, a number, symbol, array, or flavor instance. It is the item returned when you use the **car** function on a cons.

cdr
: This is the second element of a cons. It is the next cons in a list, or the symbol **nil**, representing the end of the list. However, it can also be any other Lisp object, 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 cons.

circular list
: A *circular list* is like a list, except that the *cdr* of the last cons, instead of being **nil**, is another cons from 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 valid Lisp objects, and there are functions to deal with them, many other functions have trouble with them. Functions that expect lists as their arguments often iterate down the chain of conses, waiting to see a **nil**; when handed a circular list, they compute forever. The ***print-circle*** variable is useful for printing circular lists. When the value of this variable is set to **nil**, the printing process proceeds by recursive descent. When the value is non-**nil**, the printer uses #*n*= and #*n*# syntax to indicate the circularities.

dotted list
: A *dotted list* is like a list, 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**. In a dotted list such as **(a b . c)**, the car is the symbol **a** and the *cdr* is the dotted pair **(b . c)**.

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 *values* or *properties*). For example:

```
(color red flavor hot container-type bottle)
```

Indicators cannot be duplicated, since a property list can only have one property at a time with a given name. A property list is represented as an even-numbered list of elements. For more information about plists: See the section "Property Lists".

set
: *Set* is a logical term that refers to a one-dimensional list with no repetitions. Therefore, both the lists **(a b c)** and **(a (b c) d)** are sets. The list **(a b a c)** would not be a set, since one character is repeated. There are functions that allow a list of items to be treated as a set, for example, functions to add, remove, and search for specific items in a list based on various criteria.

tree
: A *tree* is any data structure made up of conses whose cars and cdrs are other conses. At the bottom of a tree, the cars and cdrs can be any Lisp object, not only to conses. Another way of

looking at a tree is as a list of lists. For example:

```
((a . b) . (c . d))
```

Note that lists, dotted lists, trees, association lists, property lists, and circular lists are not mutually exclusive data structure types; they are different ways of looking at structures composed of conses.

Printed Representation of Lists

The printer could print all conses in the dotted form **(***car* **.** *cdr***)**, but since lists are a common type in Lisp, there is a compact format for printing them. The printed representation of a cons favors the list representation over the dotted representation.

When the printer begins to print a list, it first sees a cons. The printer has no way of telling whether a list or a tree is going to be printed. **print** starts by printing an open-parenthesis. Then it prints the car of the cons and examines the cdr of the cons. If the cdr is a cons, the printer prints a space, using this new open-parenthesis and this new cons. If the cdr is anything other than a cons or **nil**, **print** prints "space dot space", followed by that object, followed by a close-parenthesis. If the cdr is **nil**, it prints a close-parenthesis. When the car and cdr are printed, the printer recurses to the initial 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 typical printed representations such as **(a b (foo bar) c)** are produced.

To print or write the printed representation of a list or tree, you can use the functions **print** and **write**. **print** returns a specified object, for example:

```
(print '(a b c)) =>
(A B C)
(A B C)
```

write returns a specified object, for example:

```
(write '(a b c)) => (A B C)
(A B C)
```

To prevent the printed representation of a cons from growing to an unmanageable length, or depth of recursion, when printing lists the print function keeps track of the length and the depth of recursion of a list as it prints it and limits them.

The number of list elements printed is controlled by the value of the variable ***print-length***. If the list length exceeds the value of ***print-length***, **print** terminates the printed representation of the list with an ellipsis (three periods) and a close-parenthesis. For example:

```
(setq list '(a b (c) (d (e f) g))) => (A B (C) (D (E F) G))

(let ((*print-length* 2))
       (print list) nil) => (A B ...) NIL
```

If the value of the variable ***print-length*** is **nil**, or is equal to, or greater than, the number of elements in the list, ***print-length*** prints the entire list:

```
(setq list '(a b (c) (d (e f) g))) => (A B (C) (D (E F) G))

(let ((*print-length* nil))
        (print list) nil) => (A B (C) (D (E F) G)) NIL

(setq list '(a b (c) (d (e f) g))) => (A B (C) (D (E F) G))

(let ((*print-length* 6))
        (print list) nil) => (A B (C) (D (E F) G)) NIL
```

The depth of recursion printed is controlled by the value of the variable ***print-level***.

If the depth of recursion exceeds the value of ***print-level***, the portion of the list beyond the specified depth is printed as "#". For example:

```
(setq list '(a (b c) (d (e f) g))) => (A (B C) (D (E F) G))

(let ((*print-level* 2))
        (print list) nil) =>  (A (B C) (D # G)) NIL
```

If the value of the variable ***print-level*** is **nil**, or is equal to or greater than the depth of recursion, ***print-level*** prints the entire list:

```
(setq list '(a (b c) (d (e f) g))) => (A (B C) (D (E F) G))

(let ((*print-level* nil))
        (print list) nil) => (A (B C) (D (E F) G)) NIL

(setq list '(a (b c) (d (e f) g))) => (A (B C) (D (E F) G))

(let ((*print-level* 4))
        (print list) nil) =>  (A (B C) (D (E F) G)) NIL
```

These two features allow an abbreviated printing, which is more concise and suppresses detail. Of course, neither the ellipsis nor the "#" can be interpreted by **read**, since the relevant information is lost.

In general, **print** tries to print conses so that **read** can read them.

Zetalisp Note: The printing functions no longer use **zl:prinlevel** and **zl:prinlength** to control printing.

How the Reader Recognizes Lists

When the reader sees an open parenthesis, it knows that the printed representation of a cons is starting. The reader reads the next object as the car. If the next token is a single dot, the reader reads the token following the dot as a cdr, and expects it to be followed with a closed parenthesis. If the next token is not a dot,

the reader builds a list, making the cdr of this cons the cons it gets by recursing to the initial cons.

The dot that separates the two elements of a dotted-pair printed representation for a cons, for example (a . b), is only recognized if it is surrounded by delimiters (typically spaces). Thus, a dot can be freely used in other contexts, for example within print-names of symbols and within numbers.

Zetalisp Note:

1. Tokens that consist of more than one dot, but no other characters, are valid symbols in Zetalisp but errors in Common Lisp.

2. The circle-X (⊗) character is read as an octal escape: The next three characters are read and interpreted as an octal number which is, in turn, interpreted as the character whose character code is that number. This character is always taken to be an alphabetic character, just as if it had been preceded by a slash. Thus, circle-X can be used to include unusual characters in the input.

Special Types of Lists

Conses are the building blocks for several types of more complex lists. Two of these are special, in the sense that Lisp contains a number of functions intended to operate specifically on them. These are *property lists* and *association lists*.

Property Lists

Lisp has another kind of tabular data structure called a *property list* (*plist* for short) in which each of the items has some property associated with it. The implementation of a property list is a memory cell containing a list with an even number (possibly zero) of elements. Usually this memory cell is the property-list cell of a symbol, but any memory cell acceptable to **setf** can be used if **getf** and **remf** are used to manipulate it. (The functions **get** and **remprop** provide a shorthand notation for manipulating a property list referenced by the memory cell of a symbol.) In each pair of elements, the first of the pair is a keyword symbol called the *indicator* and the second is a Lisp object called the *value* (or sometimes the *property*). The elements of a property list are always processed pair-wise.

A property list looks like this:

```
(indicator value indicator value indicator value)
```

Note that the term "property list" refers to the property list itself, rather than the list of entries inside the property list.

Here is an example of a property list with actual indicators and values:

```
(manufacturer vw model gti color black miles 15000)
```

See figure ! for a cons representation of this list:

Duplication of indicators is not allowed: A property list can have only one indicator

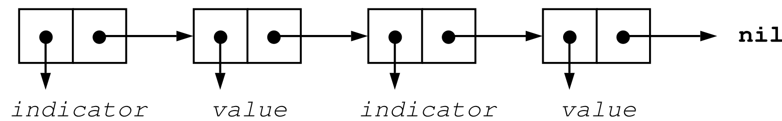


Figure 7. The Cons Representation of a (Property) List

at a time with a given name.

The kinds of operations that might be performed on property lists are adding and removing properties, and finding a property, given an item. 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.

Zetalisp Note:

Symbolics recommends that you avoid the use of disembodied property lists in new code. The documentation below is provided only to help you read old code.

Symbolics Lisp (Zetalisp, and other older Lisp dialects) do not provide for the representation of property lists by ordinary lists. Instead, older Lisp dialects provide the ability to pass around property lists independent of symbols. In order to delete elements from property lists with **zl:remprop**, it is necessary to have not just a property list, but also an object with a cell pointing to the property list so that the result, after deletion, can be stored back.

Zetalisp and the Zetalisp property list functions generalize this to allow the property list to live in any cell by means of locatives.

A typical way to construct a disembodied property list is to make a list whose first element is anything and whose cdr is the property list.

```
(setq a '(foo :a b :c d :e f)) => (FOO :A B :C D :E F)
```

List-style disembodied property lists fit into this model because the functions **get**, **getl**, **putprop**, and **remprop** operate on a *locative* as well as a symbol. Given a symbol, these functions manipulate the property list stored in a symbol's property list cell. Given a locative, they manipulate the property list referenced by **(location-contents** *locative***)**. This usage is called "disembodied property lists."

To get a locative to a location, use the function **locf**:

```
(setq a-plist (locf (cdr a))) => (FOO:A B :C D :E F)
```

Note that the result is *not* a locative. Due to considerations of cdr-coding, there is no unique location for the cdr of a cons. Instead, all the primitives that manipulate locatives treat a cons as a locative to its cdr. Thus, we can use this list as a locative and write:

```
(zl:get a-plist :c) => :D
```

When reading old code, then, you may see the code use the list directly, without bothering with the **(locf (cdr list))** construct.

To see locatives in active use, consider a property list stored in a **defstruct** slot:

```
(defstruct b-struct b) => B-STRUCT

(setq b-1 (make-b-struct :b '(:a fred :q q? :alfred hitchcock))) =>
#S(B-STRUCT :B NIL)

(setq b-plist (locf (b-struct-b b-1))) => #<DTP-LOCATIVE 60326752>

(zl:get b-plist :q) => Q?
```

Note that the **:init** keyword message protocol in the flavor system uses a locative to a property list to pass the flavor init keywords to the methods. Instead of using **:get** though, you can reference the property list with Symbolics Common Lisp functions by using the **location-contents** function to get an ordinary property list. Then you can use **getf** to retrieve individual values.

getf takes a **setf**'able reference rather than an object. Reference serves as a place to store back the modified result. Continuing the example above:

```
(setq b-modern-plist (location-contents b-plist)) =>
(:A FRED :Q Q? :ALFRED HITCHCOCK)

(getf b-modern-plist :alfred) => HITCHCOCK
```

Creating and Modifying Property Lists

In general, you can perform the same operations on the property lists of symbols as on those that are referenced by an arbitrary **setf**, but the function names differ depending on the type of property list. Here is a table showing the differences:

Operation on Property List	*For Property List of Symbol, use*	*For Generalized Property List, use*
Create/expand	**setf** with **get**	**setf** with **getf**
Access a value	**get**	**getf**
Remove a value	**remprop**	**remf**
Display	**symbol-plist**	**symbol-value**

You can use any of the property list manipulating functions on property lists created with **defvar**. For a summary of all such functions: See the section "Functions That Operate on Property Lists".

We use property lists of symbols to illustrate the various creation and manipulation operations.

A symbol is an object that has room for five components: a print name, a value binding, a property list, a function binding, and a package. When a symbol is created, its property list is **nil**. For example:

```
(defvar *colors*) => *COLORS*
```

For more information about property lists of symbols: See the section "The Property List of a Symbol".

You can also use **defvar** to create a property list with several indicator and value pairs.

```
(defvar *city* '(name portland state maine size 100000)) => *CITY*

(symbol-value '*city*) => name portland state maine size 100000)) => *CITY*
```

To associate a property list with a symbol, you can use **setf** with **get**. This limits you to creating an initial property list with only one pair of elements.

```
(setf (get 'artist 'name) 'monet) => MONET

(symbol-plist 'artist) => (NAME MONET)
```

You can associate other indicator-value pairs to a symbol's property list by repeated use of **setf** with **get**.

```
(symbol-plist 'artist) => (NAME PICASSO)

(setf (get 'artist 'style) 'cubism) => CUBISM

(symbol-plist 'artist) => (NAME PICASSO STYLE CUBISM)

(setf (get 'artist 'nationality) 'nil) => ARTIST

(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO STYLE CUBISM)
```

As the last example shows, an indictor can have a value of **nil**. You can also use **setf** and **get** to replace an old indicator-value pair with a new value. Changes to property values are destructive; once a change is made to the property list it is permanent, and the former indicator and value pair is gone.

To change the value of the indicator `style`:

```
(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO STYLE CUBISM)

(setf (get 'artist 'style) 'expressionism)) => EXPRESSIONISM

(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO STYLE EXPRESSIONISM)
```

You can remove values from a property list using the function **remprop**. This function destructively removes an indicator and its value from the property list:

```
(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO STYLE EXPRESSIONISM)
```

```
(remprop 'artist 'style) => T

(symbol-value 'artist) => (NATIONALITY NIL NAME PICASSO)
```

You can retrieve values from a property list using **get**. **get** searches the property list for an indicator that is **eq** to the indicator sought, and returns the value corresponding to that indicator. For example:

```
(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO)

(get 'artist 'name) => PICASSO
```

Note: get returns **nil** if it cannot find the requested indicator, or if the indicator found has a value of **nil**.

In the property list `artist`, the indicator `medium` does not exist:

```
(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO)

(get 'artist 'medium) => NIL
```

In the property list `artist`, the indicator `nationality` exists, but its value is **nil**:

```
(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO)

(get 'artist 'nationality) => NIL
```

Alternately, you can specify a message to be returned instead of **nil**. For example:

```
(symbol-plist 'artist) => (NATIONALITY NIL NAME PICASSO)

(get 'artist 'medium "indicator is absent, or has a value of nil") =>
"INDICATOR IS ABSENT, OR HAS A VALUE OF NIL"
```

You can display the contents of a symbol's property list with the function **symbol-plist**. Note that this function does not return the property list itself; you cannot do **get** on it. You must give the symbol itself to **get**, or use **getf**.

```
(symbol-plist 'artist)  => (NATIONALITY NIL NAME PICASSO)
```

Note that if you use **symbol-plist** with **setf** you can destructively replace the entire property list of a symbol. This is a dangerous operation that should be used with care since other applications may be sharing the property list with you.

Here are examples to create and manipulate property lists referenced by an arbitrary **setf**. As already stated, the operations themselves are analogous to those used to manipulate the property list of symbols, the only difference being in the function names. The restriction is that the *place* or property list argument of these functions be acceptable to **setf**.

To associate a property list with a symbol, use **defvar**:

```
(defvar horse nil) => HORSE
```

You can associate indicator-value pairs to a property list by repeated use of **setf** with **getf**.

```
(setf (getf horse 'color) 'brown) => BROWN

(symbol-value 'horse) => (COLOR BROWN)

(setf (getf horse 'hair) 'short)  => SHORT

(symbol-value 'horse) => (HAIR SHORT COLOR BROWN)
```

To replace a property in a property list use **setf** and **getf**:

```
(symbol-value 'horse) => (HAIR SHORT COLOR BROWN)

(setf (getf horse 'color) 'black)  => BLACK

(symbol-value 'horse) => (HAIR SHORT COLOR BLACK)
```

To destructively remove a property from a property list use **remf**:

```
(symbol-value 'horse) => (HAIR SHORT COLOR BLACK)

(remf horses 'hair) =>  T

(symbol-value 'horse) => (COLOR BLACK)
```

To retrieve a value, given an indicator, from a property list, use the function **getf**:

```
(symbol-value 'horse) => (COLOR BLACK)


(getf horse 'color) => BLACK
```

To display a property list, use the function **symbol-value**, which returns the current value of a symbol.

```
(symbol-value 'horse) => (COLOR BLACK)
```

Functions That Operate on Property Lists

The following functions add to, modify, or search property lists.

All of these functions use **eq** as the test.

defprop *sym value indicator* Gives *sym*'s property list an *indicator*-property corresponding to *value*. **defprop** is a special form.

defprop is a Symbolics extension to Common Lisp.

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.

getf *plist indicator* — Searches for the property *indicator* on *plist*.

get-properties *plist indicator-list* — Searches for a property (of *indicator-list*) on *plist*. **get-properties** returns three values. If none of the indicators is found, all three values are **nil**. If the search is successful, the first two values are the property found and its value and the third value is the tail of the property list whose car is the property found.

remprop *symbol indicator* — Removes *symbol*'s *indicator* property, by slicing it out of the property list. If the property list is associated with a symbol, use **remf**.

remf *place indicator* — Removes *indicator* from the property list stored in *place*. If it cannot find *indicator*, it returns **nil**. If the property list is associated with a symbol, use **remprop**.

symbol-plist *sym* — Returns the property list of *sym*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:get *symbol indicator* — Looks up *indicator* on *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's* property list for any of the indicators in *indicator-list* until it finds a property whose indicator is one of the elements of *indicator-list*.

zl:plist *sym* — Returns the property of *sym*.

zl:putprop *sym value indicator* — Gives *sym* an *indicator*-property of *value*.

zl:setplist *sym list* — Sets the property list of *sym* to *list*.

zl:remprop *sym indicator* — Removes *sym*'s *indicator* property, by slicing it out of the property list. It returns that portion of the list inside *sym* of which the form *indicator*-property was the car.

Note: You can do property list operations on flavor instances by including the mixin flavor **sys:property-list-mixin** in the definition of the flavor. For information on the methods provided by that mixin flavor, see the section "Property List Methods".

Dotted Lists

A *dotted list* is a special case of a list. It is not a true list, since it does not terminate in **nil**, but it is more like a list than any other data structure type. A dotted list is one whose last cons does not have **nil** for its cdr, but has some other data object (which is also not a cons) as its cdr. A dotted list looks like this:

((a . b) (c . d))

It is called "dotted" because of its special notation, that is, a left parenthesis, the printed representation of the car of the cons, a space, a period, a space, the printed representation of the cdr of the cons, and a right parenthesis. See figure ! for the cons representation of a dotted list.

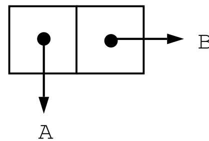


Figure 8. A Dotted List

Association Lists

Conses are the building blocks for a more complicated structure called an *association list* or *alist*.

This structure is a list of pairs (or conses) in which each pair is an association. The car of each pair is the *key* (or indicator), and the cdr is the *datum* (the value associated with that key).

An association list looks like this:

```
((indicator . value) (indicator . value) (indicator . value))
```

Here is an example of an association list with actual indicators and values:

```
((dog . poodle) (cat . coon) (bird . parrot))
```

See figure ! for the cons representation of an association list:

It is permissible for **nil** to be an element of an association list in place of an indicator and value pair.

An indicator or value can appear more than once in an association list. Duplications are allowable, since the function that adds elements to an association list always adds to the front of the list, and the function used for searching an association list always finds the first instance of a cons whose car matches the indicator.

Creating and Modifying Association Lists

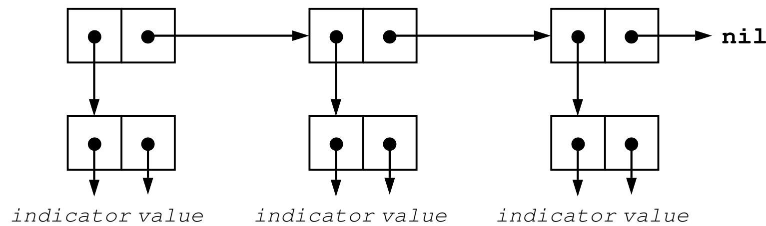


Figure 9. The Cons Representation of an Association List

You can create an association list using **setq** with **acons**. In this example, the association list gems is created with an initial indicator and value pair (jade and green):

```
(setq gems (acons 'jade 'green nil))
=> ((JADE . GREEN))
```

Using **defvar**, you can create an association list with a number of elements (indicator and value pairs) at one time.

```
(defvar *flowers* '((rose . red) (mum . yellow) (lily . white)))
=> ((ROSE . RED) (MUM . YELLOW) (LILY . WHITE))
```

Sometimes you might want to create an initial association list with all of its elements in place. At other times you might not know what the elements in the association list will be, so you might initially want to create an empty list. To create an empty association list, use **defvar**:

```
(defvar *flowers* nil)
=> *FLOWERS*
```

This allows you to create an association list before you put any elements into it.

An advantage of association lists is that they can be expanded simply by adding new entries to the front; that is, adding new indicator-value pairs is a nondestructive activity.

To expand the association list gems, for example, we can add one indicator-value pair at a time, using **setq** and **acons**:

```
(symbol-value 'gems)
=> ((JADE . GREEN))

(setq gems (acons 'onyx 'black gems))
=> ((ONYX . BLACK) (JADE . GREEN))

(setq gems (acons 'ruby 'red gems))
=> ((RUBY . RED) (ONYX . BLACK) (JADE . GREEN))
```

```
(setq gems (acons 'jade 'black gems))
=> ((JADE . BLACK) (RUBY . RED) (ONYX . BLACK) (JADE . GREEN))
```

In this last expansion we've modified the value of an existing indicator, `jade`; the resulting list contains the duplicated indicators.

It is also possible to create or expand an association list by pairing elements from two lists, using **pairlis**. For example:

```
(pairlis '(one two) '(1 2))
=> ((ONE . 1) (TWO . 2))
```

You can use the function **assoc** to retrieve pairs of indicator and value associations from a list. **assoc** searches the association list and returns the value of the first pair in the association list whose car satisfies the predicate specified by **:test**, or **nil** if no such pair is found. **eql** is the default value of **:test**. **assoc** returns both the indicator and the value (that is, the entire cons cell). To find the association between the indicator `ruby` and its value in the association list called `gems`:

```
(symbol-value 'gems)
=> ((JADE . BLACK) (RUBY . RED) (ONYX . BLACK) (JADE . GREEN))

(assoc 'ruby gems))
=> (RUBY . RED)
```

In some cases, it is desirable to regard an association list as mapping in the reverse direction, that is, mapping *from* a value *to* an indicator. The function **rassoc** is useful for searching a list using this mapping. It does a reverse association and gets the indicator given the value. To find the association between the value `red` and its indicator in the list called `gems`:

```
(symbol-value 'gems)
=> ((JADE . BLACK) (RUBY . RED) (ONYX . BLACK) (JADE . GREEN))

(rassoc 'red gems))
=> (RUBY . RED)
```

You can use the generalized sequence function **remove** to remove an indicator and value pair from an association list. **remove** finds the first instance of a cons whose car is **eql** to the indicator, and removes the pair from the association list. This function is non-destructive (the removal is not permanent) as the returned sequence is a copy of the sequence, save that some elements are not copied. Elements that are not removed occur in the same order in the result as they did in the original sequence.

For example, to remove the indicator-value pair `ruby` and `red`:

```
(symbol-value 'gems)
=> ((JADE . BLACK) (RUBY . RED) (ONYX . BLACK) (JADE . GREEN))

(remove 'ruby gems :key #'car)
=> ((JADE . BLACK) (ONYX . BLACK) (JADE . GREEN))
```

remove takes the keyword **:test**, which tests the elements according to a specified criterion. For example, suppose a list contains both an indicator and a value with the same symbol name. In order to remove the right indictor-value pair from the list, you can use the **:test** and **:key** keywords. **:test** provides the criterion to test the element against, and **:key** specifies the position of the indicator to remove. For example:

```
(symbol-value 'letters)
=> ((a .b) (b . d) (c . f))

(remove 'b letters :test #'eql :key #'car)
=> ((c. f) (a . b))
```

Note that only the indicator-value pair where `b` is the car is removed.

To make this modification permanent, we must change the value of the symbol `gems`, using the functions **assoc** and **setq** in addition to **remove**:

```
(setq gems (remove (assoc 'ruby gems) gems)
=> ((JADE . BLACK) (ONYX . BLACK) (JADE . GREEN))
```

Functions that Operate on Association Lists

All of the Common Lisp functions below use **eql** as the test.

The first two functions are used to construct association lists. The remainder are used to extract a cons pair, or list of pairs, from an association list, in accordance with some specified test. The generalized sequence function **remove** excises indicator and value pairs from the association list.

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 associates elements of the first list to corresponding elements of the second list, creating an association list.

assoc *item a-list* &key *(:test* **#'eql***) :test-not (:key* **#'identity***)*

Searches the association list *a-list*. The value returned is the first pair in *a-list* whose car satisfies the predicate specified by **:test**, or **nil** if no such pair is found.

assoc-if *predicate a-list* &key *:key*

Searches the association list *a-list*. Returns the first pair in *a-list* whose car satisfies *predicate*, or **nil** if there is no such pair in *a-list*.

assoc-if-not *predicate a-list* &key *:key*

Searches the association list *a-list*. The value returned is the first pair in *a-list* whose car does not satisfy *predicate*, or **nil** if there is no such pair in *a-list*.

rassoc *item a-list* &key *(:test* **#'eql***) :test-not (:key* **#'identity***)*
Searches the association list *a-list*. Returns the first pair in *a-list* whose cdr satisfies the predicate specified by **:test**.

rassoc-if *predicate a-list* &key *:key*
Searches the association list *a-list*. Returns the first pair in *a-list* whose cdr satisfies *predicate*.

rassoc-if-not *predicate a-list* &key *:key*
Searches the association list *a-list*. The value returned is the first pair in *a-list* whose cdr does not satisfy *predicate*.

Note: The following Zetalisp functions are included to help you read old programs.

In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:assoc *item in-list*
Looks up *item* in the association list *in-list*.

zl:ass *pred item list*
Looks up *item* in the association list *list*.

zl:assq *item list* Looks up *item* in the association list *in-list*.

zl:memass *pred item list*
Looks up *item* in the association list *list*.

zl:pairlis *vars vals* Takes two lists and makes an association list which associates elements of the first list with corresponding elements of the second list.

zl:rass *pred item in-list*
Looks up *item* in the association list *in-list*.

zl:rassoc *item in-list*
Looks up *item* in the association list *in-list*.

zl:rassq *item in-list*
Looks up *item* in the association list *in-list*.

zl:sassoc *item in-list else*
Looks up *item* in the association list *in-list*.

zl:sassq *item in-list else*
Looks up *item* in the association list *in-list*.

Trees

You can build data structures other than lists out of conses. In general, these are called trees. 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. See figure ! for the cons representation of the tree ((a . b) . (c . d))

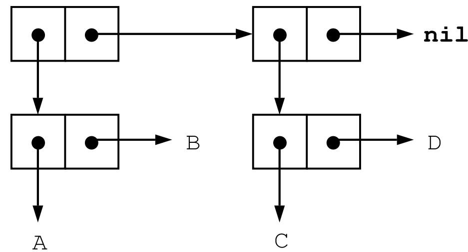


Figure 10. The Cons Representation of A Tree

See figure ! for a diagram of a tree.

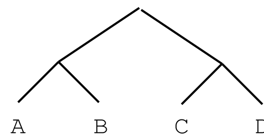


Figure 11. Diagram of a Tree

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 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

Controlling List Operations with Keyword Arguments

Some functions that operate on lists let you specify the portion of the list to be operated on. Such functions have keyword arguments **:start** and **:end**, which must be non-negative integers as follows:

> **:start** indicates the position for beginning an 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, you can specify the direction to search through the list by using the keyword **:from-end**. Where **:from-end** is present, the function normally processes the list 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 which area the object should be created in. See the section "Areas". **: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 in the list. The keyword **:initial-element** (or in Zetalisp, **:initial-value**) can be used for 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.

You can use **:test** to specify a binary operation to be applied to an argument, and each of the elements of the target list, in turn. If you do not supply **:test**, the default matching 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*.

To reverse the sense of **:test** you can use **:test-not**. 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** should be one of the following:

- **nil**
- A function of one argument that extracts from an element the part to be tested in place of the whole element.

Note that operations that test elements include 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 scenarios, a *target* element 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 the following is true:

  ```
  (funcall test-function target (funcall key-function item))
  ```

- A basic function was called, *test-function* was specified by **:test-not**, *key-function* was specified by **:key**, and the following is false:

  ```
  (funcall test-function target (funcall key-function item))
  ```

- An **-if** function was called, and the following is true:

  ```
  (funcall predicate (funcall key-function item))
  ```

- An **-if-not** function was called, and the following is false:

  ```
  (funcall predicate (funcall key-function item))
  ```

Predicates that Operate on Lists

Two groups of predicate functions operate on lists. The first group 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 *object* is not a cons, otherwise **nil**.

consp *object* — Returns **t** if *object* is a cons, otherwise **nil**.

endp *object* Tests for the end of a list. Returns **nil** when applied to a cons, and **t** when applied to **nil**.

every *predicate* &rest *sequences*
Tests each element in *sequences* against *predicate*. Returns **nil** as soon as any element fails to satisfy the test of *predicate*. Otherwise returns non-**nil**.

listp *object* Returns **t** if *object* is a cons or the empty list (), otherwise **nil**. **listp** returns **nil** if *object* is a dotted list, since it only looks at the first cons, not the last cons of a list.

nlistp *x* Returns **t** if *x* is not a cons, otherwise **nil**. Equivalent to **atom**. **nlistp** is a Symbolics extension to Common Lisp.

some *predicate* &rest *sequences*
Tests each element in *sequences* against *predicate*. Returns whatever value *predicate* returns as non-**nil**, as soon as any element satisfies the test of *predicate*. Otherwise returns **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**.

tailp *tail list* Returns **t** if *tail* is an ending sublist of *list*, 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 returns **nil**.

Note: The following Zetalisp predicates are included to help you read old programs. In your new programs, where possible, *use the Common Lisp versions of these predicates.*

zl:listp *object* Returns **t** if its argument is a cons or *not* the empty list (), otherwise **nil**. Note that **listp** and **zl:listp** are not equivalent.

zl:some *list predicate* &optional (step **#'cdr**)
Tests each element in *list* against *predicate*. Returns the tail of the list. Otherwise returns **nil**.

zl:every *list pred* &optional (*step #'*Each)
Tests each element in *list* against the *pred*. Extraction from the list can be changed by the *step* function. Returns **t** if *pred* returns non-**nil** when applied to every element of list, otherwise **nil** if predicate returns **nil** for some element.

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* Returns the number of elements in *sequence* as a non-negative integer. *sequence* can be either a list or a vector (one-dimensional array).

list-length *list* Returns, as an integer, the length of *list*. **list-length** differs from **length** when *list* is circular.

position *item sequence* &key *(:test* **#'eql***) :test-not (:key* **#'identity***) :from-end (:start* **0***) :end*

If *sequence* contains an element satisfying the predicate specified by the **:test** keyword, then **position** returns the index within the sequence of the leftmost such element as a non-negative integer; otherwise **nil** is returned. *sequence* can be either a list or a vector (one-dimensional array).

position-if *predicate sequence* &key *:key :from-end (:start* **0***) :end*

If *sequence* contains an element satisfying *predicate*, then **position** returns the index within the sequence of the leftmost such element as a non-negative integer; otherwise **nil** is returned. *sequence* can be either a list or a vector (one-dimensional array).

position-if-not *predicate sequence* &key *:key :from-end (:start* **0***) :end*

If *sequence* contains an element that does not satisfy *predicate*, then **position** returns the index within the sequence of the leftmost such element as a non-negative integer; otherwise **nil** is returned. *sequence* can be either a list or a vector (one-dimensional array).

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:length *x* Counts the number of elements in *x*. Returns a non-negative integer. The Symbolics Common Lisp equivalent of this function is **length**.

zl:find-position-in-list *item list*

Looks down *list* for an element that is **eq** to *item* and returns the numeric index of the first element that is **eq** to *item*. Returns **nil** if it does not find one.

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*.

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 (*cdr-coding* is the internal data format used to store conses inside a Symbolics computer.) For more information, see the section "Cdr-Coding".

Whenever you create a new object, you can also specify an *area* of virtual memory, with the keyword **:area**. An area is a location in virtual memory where objects and their references (or more generally, any pieces of related information), can be located near each other, that is, located at nearby addresses in virtual memory. When this is true, the paging system can avoid *thrashing*: swapping many pages in and out of main memory in order to access relatively few data.

For more background information about areas, see the section "Areas".

cons *x y* — Adds a new element to the front of a list. Returns the new cons. It is the primitive function to create a new cons whose car is *x* and whose cdr is *y*.

ncons *x* — Creates a cons whose car is *x* and whose cdr is **nil**. **ncons** is a Symbolics extension to Common Lisp.

xcons *x y* — Creates an *exchanged cons*, one whose car is *y* and whose cdr is *x*. **xcons** is a Symbolics extension to Common Lisp.

cons-in-area *x y area* — Creates a cons in a specific area.

ncons-in-area *x area* — Creates a cons with a car of *x* in a specific area. **ncons-in-area** is a Symbolics extension to Common Lisp.

xcons-in-area *x y area* — Creates an exchanged cons in a specific area. **xcons-in-area** is a Symbolics extension to Common Lisp.

list &rest *elements* — Constructs and returns a list of its arguments.

list* &rest *args* — Constructs a list whose last cons is "dotted." Takes at least one argument.

list-in-area *area* &rest *elements* — Same as **list**, except that it takes an area argument, and creates the list in that area. **list-in-area** is a Symbolics extension to Common Lisp.

list*-in-area *area* &rest *args* — Same as **list***, except that it takes an area argument, and creates the list in that area. **list*-**

in-area is a Symbolics extension to Common Lisp.

make-list *size* &key *:initial-element :area*
Creates and returns a list containing *size* elements. This function has the optional argument *area*, which is a Symbolics extension to Common Lisp.

circular-list &rest *args*
Constructs a circular list whose elements are *args*, repeated infinitely. Often used with mapping. **circular-list** returns a list whose last cdr is is a list, instead of **nil**. **circular-list** is a Symbolics extension to Common Lisp.

nconc &rest *arg*
Takes lists as arguments. Destructively concatenates and returns *arg*s in a list. See the function **concatenate**.

nreconc *l tail*
Creates a list that is the first argument reversed concatenated with the second argument.

append &rest *lists*
Concatenates the *lists*, returning the resulting list.

revappend *x y*
Concatenates *x* and *y*, returning the resulting list in reverse order.

adjoin *item list* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) (*:area* **sys:default-cons-area**) *:localize :replace*
Adds *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 *item* onto the top of the stack. This function has the optional argument *area*, which is a Symbolics extension to Common Lisp.

pushnew *item reference* &key *:test :test-not :key :area :localize :replace*
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.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:make-list *length* &key *area initial-value*
Creates and returns a list containing *length* elements. Use the Common Lisp function **make-list**.

zl:push *item list*
Adds *item* to the front of *list*, which should be stored in a generalized variable. Use the Common Lisp function **push**.

zl:push-in-area *item list area*
Adds *item* to the front of *list*, which should be stored in a generalized variable.

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 *list* &optional *area force-dotted*
Makes a copy of *list* that is **equal** to *list*, but not **eq**. (Only the top level of list structure is copied). **copy-list** can be used to convert a list into compact, cdr-coded form. This function has the optional argument *area*, which is a Symbolics extension to Common Lisp.

copy-list* *list* &optional *area*
Same as **copy-list**, except that the last cons of the resulting list is never cdr-coded. **copy-list*** is a Symbolics extension to Common Lisp.

copy-alist *al* &optional *area*
Makes a copy of the association list *al* that is **equal** to *al*, but not **eq** (only the two top levels of list structure are copied). This function has the optional argument *area*, which is a Symbolics extension to Common Lisp.

copy-tree *tree* &optional *area*
Copies a tree of conses. This function has the optional argument *area*, which is a Symbolics extension to Common Lisp.

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:copy-if-necessary** checks whether *thing* is in a temporary area of some kind, and moves it if it is. If *thing* is not in a temporary area, it is simply returned.

sys:localize-list *list* &optional *area*
Improves locality of incrementally-constructed lists and association lists.

sys:localize-tree *tree* &optional (*n-levels 100*) *area*
Improves locality of incrementally-constructed lists and trees.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:copylist *list* &optional *area force-dotted*
Makes a copy of *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. **zl:copylist** converts a list into compact, cdr-coded form. Use the Common Lisp function **copy-list**.

zl:copylist* *list* &optional *area*
Same as **zl:copylist**, except that the last cons of the resulting list is never cdr-coded. Use the Common Lisp function equivalent to **copy-list***.

zl:copyalist *al* &optional *area*
Copies the conses, but not elements, in association list *al*. In addition, each element of *al* that is a cons is replaced in the copy by a new cons with the same car and cdr. You can optionally specify the area in which to create the new copy. The default is to copy the new list into the area occupied by the old list. Returns an association list that is **equal** to *al*, but not **eq**. Use the Common Lisp function **copy-alist**.

zl:copytree *tree* &optional *area*
Copies a tree of conses. Use the Common Lisp function **copy-tree**.

zl:copytree-share *tree* &optional *area* **(cl:make-hash-table :test #'equal :locking nil :number-of-values 0))** *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**.

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 cars and cdrs, for example, **caar** and **caddr**. This represents

	the number of levels **car**s and **cdr**s that are defined as separate functions. These **car** and **cdr** functions can represent up to four **car** and **cdr** operations.
first *list*	Returns the first element of *list*. **first** is equivalent to **car**.
second *list*	Returns the second element of *list*.
third *list*	Returns the third element of *list*.
fourth *list*	Returns the fourth element of *list*.
fifth *list*	Returns the fifth element of *list*.
sixth *list*	Returns the sixth element of *list*.
seventh *list*	Returns the seventh element of *list*.
eighth *list*	Returns the eighth element of *list*.
ninth *list*	Returns the ninth element of *list*.
tenth *list*	Returns the tenth element of *list*.
last *list*	Returns the last cons of *list*.
nleft *n l* &optional *tail*	Returns the result of taking the **cdr** of *l* enough times so that taking *n* more cdrs would yield *tail*. When *tail* is **nil**, **nleft** simply returns the last *n* elements of *list*. **nleft** is a Symbolics extension to Common Lisp.
nth *n object*	Returns the *n*th element of *object*, where the zeroth element is the car of the list.
nthcdr *n list*	Takes the **cdr** of *list n* times, and returns the result.
rest *x*	Returns the cdr of *x*. **rest** is the equivalent of **cdr** and complements **first**, as **cdr** complements **car**.
some *predicate* &rest *sequences*	Tests eache element in *sequences* against *predicate*. Returns whatever value *predicate* returns as non-**nil**, as soon as any element satisfies the test of *predicate*. Otherwise returns **nil**.
pop *list*	Returns the car of the contents of *list*, and as a side effect, the cdr of contents is stored back into *list*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:firstn *n list* — Returns a list whose elements are the first *n* elements of *list*.

zl:rest1 *list* — Returns the rest of the elements of *list*, starting with element 1 (counting the first element as the zeroth).

zl:rest2 *list* — Returns the rest of the elements of *list*, starting with element 2 (counting the first element as the zeroth).

zl:rest3 *list* — Returns the rest of the elements of *list*, starting with element 3 (counting the first element as the zeroth).

zl:rest4 *list* — Returns the rest of the elements of *list*, starting with element 4 (counting the first element as the zeroth).

Functions for Modifying Lists

This group contains functions 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 *cons* to *x* and returns (the modified) *x*.

rplacd *cons x* — Changes the cdr of *cons* to *x* and returns (the modified) *x*.

pop *list* — Returns the car of *list*, and as a side effect, the cdr is stored back into *list*.

butlast *x* &optional *(n 1)* — Creates and returns a list with the same elements as *x*, excepting the last element.

remove *item sequence* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) *:from-end* (*:start* **0**) *:end :count*

Non-destructively removes items matching *item* from *sequence*. Returns the new sequence.

delete *item sequence* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) *:from-end* (*:start* **0**) *:end :count*

Destructively removes items matching *item* from *sequence*. Returns the modified sequence.

sublis *alist tree* &rest *args* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**)

Non-destructively substitutes elements from *alist* for ojects in *tree*.

nsublis *alist tree* &rest *args* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**)
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.

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 do not satisfy *predicate*. It returns the modified copy of tree, and the original tree is unchanged, although it can share with parts of the result tree.

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**.

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 *sequence*. Returns a new, reversed sequence.

nreverse *sequence*
Destructive version of **reverse**. Returns a modified sequence.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:pop *list* &optional *dest*
Returns the car of the contents of *list*, and as a side effect, the cdr of contents is stored back into *list*.

nbutlast *list* &optional *(n 1)*
Destructive version of **butlast**.

zl:remove *item list* &optional (*ntimes most-positive-fixnum*)
Non-destructive version of **zl:delete**. Use the Common Lisp function **remove**.

zl:rem *predicate item list* &optional (*ntimes most-positive-fixnum*)
Non-destructively removes occurrences of *item* that satisfy *predicate* from *list*.

zl:delete *item list* &optional (*ntimes most-positive-fixnum*)
Deletes the first *ntimes* occurrences of *item* in *list* (**equal** is used for the comparison). Returns *list* with all occurrences of *item* removed. Use the Common Lisp function **delete**.

zl:rem-if *pred list* &rest *extra-lists* Means "remove from list 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 from list 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 *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 *list* with all occurrences of *item* removed. **eq** 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 from list if this condition is not true." **zl: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 from list if this condition is not true." **zl:subset-not** refers to the function's action if *list* is considered to represent a mathematical set. See **zl:rem-if**.

zl:sublis *alist form* Non-destructively substitutes elements from *alist* for ojects in *form*.

zl:nsublis *alist form* Destructive version of **zl:sublis**.

zl:subst *new old tree* — Substitutes *new* for all occurrences of *old* in *tree*, and returns the modified copy of *tree*.

zl:nsubst *new old s-exp* — Destructive version of **zl:subst**.

zl:reverse *list* — Reverses the elements of *list*. Returns a new reversed list.

zl:nreverse *l* — Destructive version **zl:reverse**. Returns a modified list.

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.

nunion *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Destructive version of **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.

nintersection *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Destructive version of **intersection**.

set-difference *list1 list2* &key (*test #'eql*) *test-not* (*key #'identity*)
Non-destructively returns a list of elements of *list1* that do not appear in *list2*. You can also use the sequence function **mismatch**. For information, see the section "Searching for Sequence Items".

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-destructively 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* — Returns a new list, whose elements are those elements of *list* that appear before *sublist*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:union &rest *lists* — Takes two lists and returns a new list containing everything that is an element of either of the lists, using **eq** for comparison.

zl:nunion &rest *lists* — Destructive version of **zl:union**.

zl:intersection &rest *lists* — Takes two lists and returns a new list containing everything that is an element of both lists, using **eq** for comparison.

zl:nintersection &rest *lists* — Destructive version of **zl:intersection**.

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 matches *item* according to the predicate supplied for **:test**.

member-if *predicate list* &key *key* — Searches for an element in *list* that satisfies *predicate*.

member-if-not *predicate list* &key *key*
Searches for the first element in *list* that does not satisfy *predicate*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:member *item in-list* — Searches *in-list* for an element, using **equal** for the comparison.

zl:memq *item in-list* — Returns **nil** if *item* is not one of the elements of *in-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**.

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, as long as the predicate is one that 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 **rplacd**s in the same manner as **nreverse**. Thus, if you do not want the argument affected, you must sort a copy of the argument, obtainable by **zl:fillarray** or **copy-list**, as appropriate. Furthermore, **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". 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*.

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.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:sort *x sort-lessp-predicate* — Destructively modifies *x* by sorting it according to an order determined by *sort-lessp-predicate*.

zl:stable-sort *x sort-lessp-predicate-on-car*
Same as **zl:sort**, however **zl:stable-sort** guarantees that elements considered equal by *sort-lessp-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 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.

Cdr-Coding

This section explains the internal data format used to store conses inside the Symbolics machine. It is only important to read this section if you require extra storage ef ficiency 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 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. This field can have three meaningful values: *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, a problem arises: No memory cell is storing the cdr of the cons. That is the cell that has been optimized out; it just does not exist.

This problem is resolved by the use of "invisible pointers". An invisible pointer is a special kind of pointer, recognized by its data type (Symbolics 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, that word is an invisible pointer, 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, that 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 **rplacd** is done 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, **rplacd** is used on the conses, then invisible pointers are created, extra memory is allocated, and use 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 rep-

resentation 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**.

List Functions and Macros in the CL Package with SCL Extensions

Here are the list functions and macros that have Symbolics Common Lisp extensions:

Function/Macro	*Extension(s)*
assoc-if	*:key*
assoc-if-not	*:key*
copy-alist	*area*
copy-list	*area*, *force-dotted*
copy-tree	*area*
make-list	*:area*
push	*:area*, *:localize*
pushnew	*:*area, *:localize*, *:replace*
rassoc-if	*:key*
rassoc-if-not	*:key*

Arrays

The basic concepts and terminology associated with arrays are described elsewhere: See the section "Overview of Arrays".

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**), and alter elements (**setf** used with **aref**).

There are many types of array operations. Most of these can be done with specialized array functions, while some can be done with more general-purpose sequence functions.

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.

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 ! shows the relationships among the various array types.

Basic Array Functions

Symbolics Common Lisp provides the following basic operations for 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*
Creates and returns a new array.

aref *array* &rest *subscripts*
Returns the element of *array* selected by the *subscripts*.

setf *reference value* &rest *more-pairs*
Takes a form that *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing. When used with **aref**, stores a value into the specified array element.

locf *reference* Converts *reference* to a new form that creates a locative pointer to that cell.

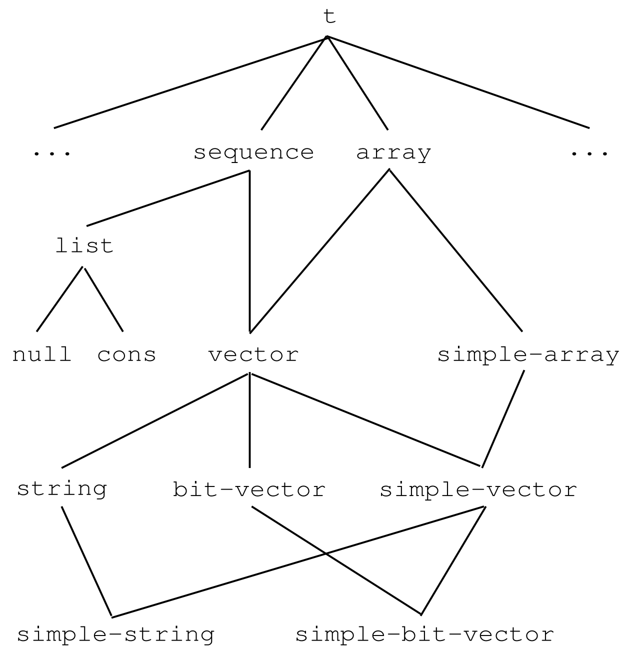


Figure 12. Symbolics Common Lisp Array Types

These constants contain implementation-specific limits on arrays:

array-rank-limit Represents the exclusive upper bound on the rank of an array.

array-dimension-limit
Represents the upper exclusive bound on each individual dimension of an array.

array-total-size-limit
Represents the exclusive upper bound on the number of elements of an array.

array-leader-length-limit
This is the exclusive upper bound of the length of an array leader.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:make-array *dimensions* &key *:area :type :displaced-to :displaced-index-offset :displaced-conformally :adjustable :leader-list :leader-length :named-structure-symbol :initial-value :fill-pointer*
Creates and returns a new array.

zl:aset *element array* &rest *subscripts*

Stores *element* into the element of *array* selected by the *subscripts*.

zl:aloc *array* &rest *subscripts*

Returns a locative pointer to the element of *array* selected by the *subscripts*. Note that the Common Lisp combination **locf** of **aref** is preferred.

For summaries of additional array operations: See the section "Common Operations on Arrays".

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*

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)
```

You can use these element types: **bit**, **string-char**, (**unsigned-byte** 8), (**unsigned-byte** 16), (**signed-byte** 8), and (**signed-byte** 16).

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.

Compatibility Note: The optional arguments **:displaced-conformally**, **:area**, **:leader-list**, **:leader-length**, and **:named-structure-symbol** are Symbolics extensions to Common Lisp, and are not available in CLOE.

For a table of related items: See the section "Basic Array Functions".

See the section "Examples of **make-array**".

If you are using CLOE, see the section "Keyword Options for **make-array**".

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 objects 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".

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 Genera does not have specialized arrays that hold just strings.

Note: The following is not the correct way to make a string:

```
(make-array 5 :element-type 'string)
```

This specifies an array whose elements are themselves strings (which is a generalized array, because Genera does not have specialized arrays that only hold strings). See the section "Strings".

: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.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:adjust-array-size
Resizes or reshapes the first dimension of an array. Use the Common Lisp function **adjust-array**.

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".

: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". See the section "Indirect Arrays".

:displaced-index-offset

If this is present, the value of the **:displaced-to** option should be an array, and the value of this should be a non-negative integer; it is made to be the index-offset of the created indirect or displaced array. See the section "Indirect Arrays".

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".

: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".

: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**, which is 1024 on Symbolics 3600-family computers and 256 on Ivory-based machines.

: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*, 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". If the array has a leader, 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, this symbol is stored in array element 0.

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.

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)
```

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.

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**.

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". 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.

Operations on Array Leaders

The following functions are available for use with arrays that have leaders:

array-has-leader-p *array*

Returns **t** if *array* has a leader; otherwise it returns **nil**.

array-leader *array index*

Returns the *index*ed element of *array*'s leader. You can use **setf** and **locf** of **array-leader**.

array-leader-length *array*

Returns the length of *array*'s leader if it has one, or **nil** if it does not.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:ap-leader *array index*

Returns a locative pointer to the *index*ed element of *array*'s leader. Use the Common Lisp combination, **locf** of **array-leader**.

zl:store-array-leader *value array index*

Stores *value* in the *index*ed element of *array*'s leader. Use the Common Lisp combination, **setf** of **array-leader**.

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's 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's elements are bytes, there is no problem. It is important to know, in this case, that the elements of such arrays are allocated from right to left within the 32-bit words.

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, you make it possible to access elements 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 **(unsigned-byte-*m*)** array to an **(unsigned-byte-*n*)** 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.

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 you use index offsets with multidimensional arrays, there is only one index offset; it is added in to the "linearized" subscript that 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)
```

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**. See the section "Rasters".

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

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, predicates such as **simple-vector-p** and **simple-bit-vector-p** can return **t** for adjustable vectors. Genera does not enforce the condition

that a simple array must not be adjustable, and, in fact, most Genera arrays are adjustable.

Bit vectors are vectors that require their elements to be of type **bit**. SCL provides functions that operate on arrays of bits (which are not constrained to be vectors): See the section "Arrays of Bits".

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".

Operations on Vectors

Symbolics Common Lisp provides the following functions for performing operations on vectors:

vector &rest *objects*
Creates a simple vector with specified initial contents.

array-has-fill-pointer-p *array*
Returns **t** if the array has a fill pointer; otherwise it returns **nil**.

fill-pointer *array* Returns the value of the fill pointer.

sys:vector-bitblt *alu size from-array from-index to-array to-index*
Copies a linear portion of *from-array* of length *size* starting at *from-index* into a linear portion of *to-array* starting at *to-index*.

vector-push *new-element vector*
Stores *new-element* in the element designated by the fill pointer and increments the fill pointer by one.

vector-push-extend *new-element vector* &optional *extension*
Stores *new-element* in the element designated by the fill pointer and increments the fill pointer by one.

vector-push-portion-extend *to-array from-array* &optional *(from-start* **0***) from-end*
Copies a portion of one array to the end of another, updating the fill pointer of the second to reflect the new contents.

vector-pop *array* &optional *default*
Decreases the fill pointer by one and returns the vector element designated by the new value of the fill pointer.

Symbolics Common Lisp provides the following predicate functions for determining

if a given object is a vector, or a specialized vector:

vectorp *object* Tests whether the given *object* is a vector.

simple-vector-p *object*
Tests whether the given *object* is a simple general vector.

bit-vector-p *object* Tests whether the given *object* is a bit vector.

simple-bit-vector-p *object*
Tests whether the given *object* is a simple bit vector.

bit-vector-zero-p *bit-vector* &key *(:start* **0**) *:end*
Tests whether the *bit vector* is a bit vector of zeros in a range specified by *:start* and *:end*.

bit-vector-cardinality *bit-vector* &key *(:start* **0**) *:end*
Tests how many of the bits in the range are one's and returns the number found.

bit-vector-position *bit bit-vector* &key *(:start* **0**) *:end*
If *bit-vector* contains an element satisfying *bit*, returns the index within the bit vector of the leftmost such element as a non-negative integer; otherwise **nil** is returned.

bit-vector-equal *bit-vector-1 bit-vector-2* &key *(:start1* **0**) *:end1 (:start2* **0**) *:end2*
Tests if two bit vectors are equal in a range specified by *:start1 :end1 :start2 :end2*.

bit-vector-subset-p *bit-vector-1 bit-vector-2* &key *(:start1* **0**) *:end1 (:start2* **0**) *:end2*
Tests if one bit-vector is a subset of another bit-vector in a range specified by *:start1 :end1 :start2 :end2*.

bit-vector-disjoint-p *bit-vector-1 bit-vector-2* &key *(:start1* **0**) *:end1 (:start2* **0**) *:end2*
Tests if two bit vectors are disjoint in a range specified by *:start1 :end1 :start2 :end2*.

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 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.

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.

For a table of related low-level raster functions: See the section "The Paging System".

bitblt *alu width height from-array from-x from-y to-array to-x to-y*
Copies a rectangular portion of *from-raster* into a rectangular portion of *to-raster*.

decode-raster-array *raster*
Returns the following attributes of the raster as values: width, height, and spanning width.

make-raster-array *width height* &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*
Makes rasters; this should be used instead of **make-array** when making arrays that are rasters.

raster-aref *raster-array x y*
Accesses the (*x*,*y*) graphics coordinate of *raster*.

raster-index-offset *raster x y*
Returns a linear index of the array element referenced by the (x,y) coordinate of the raster.

raster-width-and-height-to-make-array-dimensions *width height*
Creates an argument that can be used to call **make-array**.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp version of this function.

zl:make-raster-array *width height* &rest *make-array-options*
This function is provided for compatibility with previous releases. Use the Common Lisp function, **make-raster-array**.

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 data has been stored (and others, in general, that 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, the **zl:plane-origin** is all zeroes and you can use regular array functions, such as **aref** to access the portion of the plane that 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.

Operations on Planes

The following functions are available for using with planes:

make-plane *rank* &key *(:type* **'sys:art-q***) :default-value (:extension* **32***) :initial-dimensions :initial-origins*
Creates and returns a plane.

plane-aref *plane* &rest *point*
Returns the contents of a specified element of a plane.

plane-default *plane*
Returns the contents of the infinite number of plane elements that are not actually stored.

plane-extension *plane*
Returns the amount to extend the plane by in any direction when **zl:plane-store** is done outside of the currently stored portion.

zl:plane-aset *datum plane* &rest *point*
Stores *datum* into the specified element of a plane, extending it if necessary, and returns *datum*. Use the Common Lisp equivalent, **setf** of **plane-aref**.

zl:plane-origin *plane*
Returns a list of numbers, giving the lowest coordinate values actually stored.

zl:plane-ref *plane point*
Returns the contents of a specified element of a plane.

zl:plane-store *datum plane point*
Stores *datum* into the specified element of a plane, extending it if necessary, and returns *datum*.

Array Registers

The **aref**, **setf** of **aref**, and **zl:aset** operations on arrays consist of two parts:

1. They "decode" the array, determining type, rank, length, and the address of its first data element.

2. They read or write the requested element.

The first part of this operation does not depend 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".

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 as 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.

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**.

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.

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 need to call **math:decompose** only 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.

Operations on Matrices

The following functions perform some useful matrix operations:

math:decompose *a* &optional *lu ps ignore*
: Computes the LU decomposition of matrix *a*.

math:determinant *matrix*
: Returns the determinant of *matrix*.

math:fill-2d-array *array list*
: The opposite of **math:list-2d-array**. *list* should be a list of lists, with each element being a list corresponding to a row.

math:invert-matrix *matrix* &optional *into-matrix*
: Computes the inverse of *matrix*.

math:list-2d-array *array*
: Returns a list of lists containing the values in *array*, which must be a two-dimensional array.

math:multiply-matrices *matrix-1 matrix-2* &optional *matrix-3*
: Multiplies *matrix-1* by *matrix-2*.

math:solve *lu ps b* &optional *x*
: Takes the LU decomposition and associated permutation array produced by **math:decompose**, and solves the set of simultaneous equations defined by the original matrix *a* and the right-hand sides in the vector *b*.

math:transpose-matrix *matrix* &optional *into-matrix*
: Transposes *matrix*.

Common Operations on Arrays

Getting Information About an Array

The following functions can be used to get information about arrays:

array-dimension *array axis-number*
: Returns the length of the dimension numbered *dimension-number* of *array*.

array-dimensions *array*
: **array-dimensions** returns a list whose elements are the dimensions of *array*.

array-has-leader-p *array*
: Returns **t** if *array* has a leader; otherwise it returns **nil**.

array-in-bounds-p *array* &rest *point*
: Checks whether *subscripts* is a valid set of subscripts for *array*, and returns **t** if they are; otherwise it returns **nil**.

array-leader-length *array*
: Returns the length of *array*'s leader if it has one, or **nil** if it does not.

length *sequence*
: Returns the number of elements in *sequence* as a non-negative integer. *sequence* can be either a list or a vector (one-dimensional array).

array-rank *array*
: Returns the number of dimensions of *array*.

array-row-major-index *array* &rest *subscripts*
: Takes an array and valid subscripts for the array and returns a single positive integer, less than the total size of the array, that identifies the accessed element in the row-major ordering of the elements.

array-total-size *array*
: Returns the total number of elements in *array*.

array-element-type *array*
: Returns the type of the elements of *array*.

adjustable-array-p *array*
: Returns **t** if *array* is adjustable, and **nil** if it is not.

sys:array-row-span *array*
: Returns the number of array elements spanned by one of its rows, given a two-dimensional *array*.

sys:array-displaced-p *array*
: Tests whether the array is a displaced array.

sys:array-indexed-p *array*
: Returns **t** if *array* is an indirect array with an index-offset.

sys:array-indirect-p *array*
: Returns **t** if *array* is an indirect array.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:array-active-length *array*
Returns the number of active elements in *array*. Use the Common Lisp function, **length**.

zl:array-dimension-n *n array*
Returns the size for the specified dimension of the array. Use the Common Lisp equivalent, **array-dimension**.

zl:array-length *array*
Returns the number of elements in an array. Use **array-total-size** which is the Common Lisp equivalent of **zl:array-length**.

zl:array-#-dims *array*
Returns the dimensionality of an array. Use the Common LIsp function, **array-rank**.

Changing the Size of an Array

The following function can be used to modify the size of an existing array:

adjust-array *array new-dimensions* &key *:element-type :initial-element :initial-contents :fill-pointer :displaced-to :displaced-index-offset :displaced-conformally*
Changes the dimensions of an array. Returns an array of the same type and rank as *array*, but with the *new-dimensions*. The number of *new-dimensions* must equal the rank of the array.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:adjust-array-size *array new-index-length*
Resizes or reshapes the first dimension of an array. Use the Common Lisp function **adjust-array**.

zl:array-grow *array* &rest *dimensions*
Creates a new array of the same type as *array*, with the specified dimensions.

Arrays of Bits

The following functions are available for use with arrays of bits:

bit *array* &rest *subscripts*
Returns the element of *array* selected by the *subscripts*.

sbit *array* &rest *subscripts*
Returns the element of *array* selected by the *subscripts*.

bit-and *first second* &optional *third*
Performs logical *and* operations on bit arrays.

bit-ior *first second* &optional *third*
Performs logical *inclusive or* operations on bit arrays.

bit-xor *first second* &optional *third*
Performs logical *exclusive or* operations on bit arrays.

bit-eqv *first second* &optional *third*
Performs logical *exclusive nor* operations on bit arrays.

bit-nand *first second* &optional *third*
Performs logical *not and* operations on bit arrays.

bit-nor *first second* &optional *third*
Performs logical *not or* operations on bit arrays.

bit-not *source* &optional *destination*
Returns a bit-array of the same rank and dimensions that contains a copy of the argument with all the bits inverted.

bit-andc1 *first second* &optional *third*
Performs logical *and* operations on the complement of *first* with *second* on bit arrays.

bit-andc2 *first second* &optional *third*
Performs logical *and* operations on *first* with the complement of *second* on bit arrays

bit-orc1 *first second* &optional *third*
Performs logical *or* operations on the complement of *first* with *second* on bit arrays.

bit-orc2 *first second* &optional *third*
Performs logical *or* operations on *first* with the complement of *second* on bit arrays.

bit-vector-p *object* Tests whether the given *object* is a bit vector.

Adding to the End of an Array

The following functions can be used to add to the end of an array:

vector-pop *array* &optional *default*
Decreases the fill pointer by one and returns the vector element designated by the new value of the fill pointer.

vector-push *new-element vector*
Stores *new-element* in the element designated by the fill pointer and increments the fill pointer by one.

vector-push-extend *new-element vector* &optional *extension*

Stores *new-element* in the element designated by the fill pointer and increments the fill pointer by one.

vector-push-portion-extend *to-array from-array* &optional *(from-start* **0***) from-end*

Copies a portion of one array to the end of another, updating the fill pointer of the second to reflect the new contents.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

zl:array-pop *array* &optional *default*

Decreases the fill pointer by one. Use the Common Lisp equivalent, **vector-pop**. Use the Common Lisp function, **vector-pop**.

zl:array-push *array x*

Attempts to store *x* in the element of the array designated by the fill pointer and increase the fill pointer by one. Use the Common Lisp function, **vector-push**.

zl:array-push-extend *array data* &optional *extension*

This function is similar to **zl:array-push**, except that if the fill pointer gets too large, the array is grown to fit the new element. Use the Common Lisp function, **vector-push-extend**.

zl:array-push-portion-extend *to-array from-array* &optional *(from-start* **0***) from-end*

Copies a portion of one array to the end of another, updating the fill pointer of the other to reflect the new contents. Use the Common Lisp function, **vector-push-portion-extend**.

Copying an Array

The following functions can be used to copy the contents of arrays:

2d-array-blt *alu nrows ncolumns from-array from-row from-column to-array to-row to-column*

Copies a rectangular portion of *from-array* into a portion of *to-array*.

bitblt *alu width height from-array from-x from-y to-array to-x to-y*

Copies a rectangular portion of *from-raster* into a rectangular portion of *to-raster*.

copy-array-contents *from-array to-array*

Copies the contents of *from-array* into the contents of *to-array*, element by element.

copy-array-contents-and-leader *from-array to-array*

Copies the contents and leader of *from-array* into the contents of *to-array*, element by element.

copy-array-portion *from-array from-start from-end to-array to-start to-end*

Copies the portion of the array *from-array* with indices greater than or equal to *from-start* and less than *from-end* into the portion of the array *to-array* with indices greater than or equal to *to-start* and less than *to-end*, element by element.

list-array-leader *array* &optional *limit*

Creates and returns a list whose elements are those of *array*'s leader.

replace *sequence1 sequence2* &key *(:start1* **0***) :end1 (:start2* **0***) :end2*

Destructively modifies *sequence1* by copying into it successive elements from *sequence2*.

Converting Between Arrays and Lists

The following functions convert between arrays and lists:

zl:fillarray *array source*

Fills up *array* with the elements of *source*.

zl:listarray *array* &optional *limit*

Creates and returns a list whose elements are those of *array*.

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".

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**.

sys:%1d-aref *array i0*

Returns the element of *array* selected by the *index*.

sys:%1d-aloc *array i0*

Like **zl:aloc** except that it ignores the the number of dimensions of the array and acts as if it were a one-dimensional array.

sys:array-row-span *array*

Returns the number of array elements spanned by one of its rows, given a two-dimensional *array*.

Accessing Arrays Specially

The function **sys:array-row-span** is for users of **sys:%1d-aref** and the **sys:array-register-1d** declaration is for users who need to perform their own subscript calculations and do special loop optimizations.

sys:array-row-span *array* *Function*

Returns the number of array elements spanned by one of its rows, given a two-dimensional *array*. Normally, this is just equal to the length of a row (that is, the number of columns), but for conformally displaced arrays, the length and the span are not equal.

```
(sys:array-row-span (make-array '(4 5))) => 5
(sys:array-row-span (make-array '(4 5)
                                  :displaced-to (make-array '(8 9))
                                  :displaced-conformally t))
        => 9
```

Note: If the array is conceptually a raster, it is better to use **decode-raster-array** than **sys:array-row-span**.

For a table of related items: See the section "Getting Information About an Array". See the section "Accessing Multidimensional Arrays as One-dimensional".

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 and symbols with size.

sys:array-bits-per-element *array-type*
: A function that returns the number of bits per cell for unsigned numeric arrays.

sys:array-element-size *array*
: Given an array, returns the number of bits that fit in an element of that array.

sys:array-element-byte-size *array*
: Given an array, returns the number of bits that fit into an element of that 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 *array-type*
: A function that returns the number of array elements stored in one word.

sys:array-types *index*

Returns the symbolic name of the array type.

Other Array Functions

sys:return-array *array*

This function attempts to return *array* to free storage. This is a subtle and dangerous feature.

sys:with-stack-array *(var length* &key *:type :element-type :initial-element :initial-contents :displaced-to :displaced-index-offset :displaced-conformally :leader-list :leader-length :named-structure-symbol :initial-value :fill-pointer)* &body *body*

Like **with-stack-list**, but makes an array.

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.

Compatibility Operations for Arrays

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 **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 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 you know beforehand that the integers stored are nonnegative and limited to a certain number of bits. They occupy less storage than **sys:artq** arrays, 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, fit into one word).

sys:art-string Array Type

A **sys:art-string** array is one 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".

sys:art-boolean Array Type

A **sys:art-boolean** array is one whose elements can take on the values **t** and **nil**. It uses only one bit of storage per element.

sys:art-fixnum Array Type

A **sys:art-fixnum** array is one that stores fixnums only. It is similar to the **sys:art-1b**, 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.

Zetalisp Array-Accessing Primitives

You 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**.

Array Functions in the CL Package with SCL Extensions

Here are the array functions that have Symbolics Common Lisp extensions:

Function	*Extension(s)*
make-array	*:displaced-conformally*, *:area*, *:leader-list*, *:leader-length, :named-structure-symbol*
adjust-array	*:displaced-conformally*

Sequences

Introduction to Sequences

A *sequence* is a data 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 and vectors. These functions perform basic operations on sequences of Lisp objects, irrespective of their underlying representation. 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.

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 Lists".

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 enclosed in double quotes ("). See the section "How the Reader Recognizes Strings".

Printed Representation of Sequences

The printed representation of a list starts with an open parenthesis, as in:

```
(foo bar baz)
```

See the section "Printed Representation of Lists".

The printed representation of a vector (a one-dimensional array) is not very meaningful. It describes the symbolic type of the array, the size of the dimension, and the memory location of the array. The display begins with a pound sign and is enclosed by angle brackets, as in:

```
#<ART-Q-10 28423710>
```

Type Specifiers and Type Hierarchy for Sequences

The type specifiers relating to sequences are:

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 ! 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".

Sequence Operations

The sequence operations fall into six major categories:

- Constructing and accessing
- Predicates

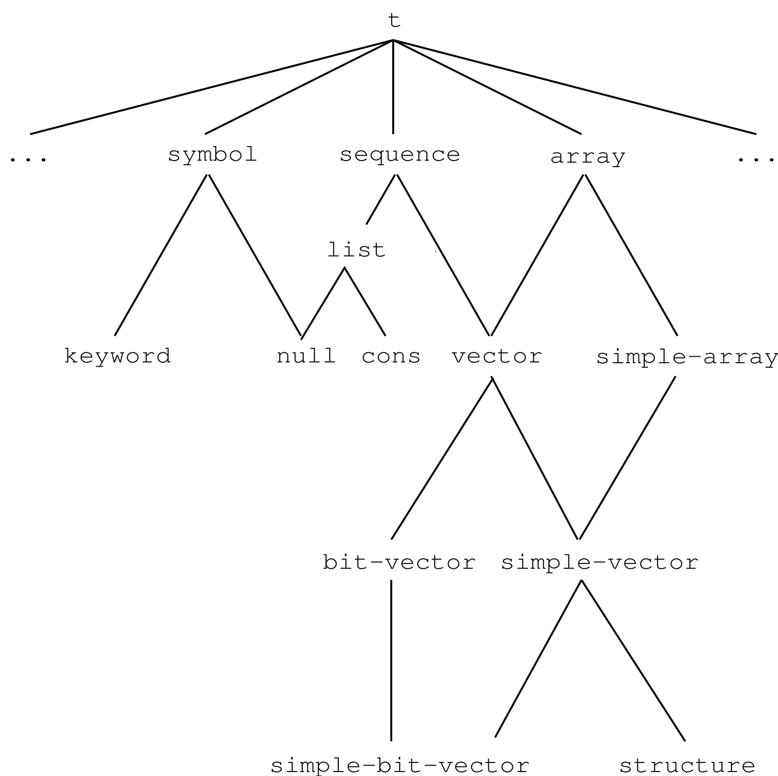


Figure 13. Symbolics Common Lisp Sequence Data Types

- Mapping
- Modifying
  - Reducing
  - Replacing
- Searching
- Sorting and merging

Whenever a sequence function constructs or returns a new vector, it always returns a simple vector; similarly, any strings constructed are simple strings.

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.

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, 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, **remove** 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-argument predicate (not an item), and sequence elements are tested for satisfying and not satisfying the predicate. For example, **remove-if** 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 below 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: **find** 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.

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 are simple strings.

elt *sequence index* — Extracts an element from *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 and returns a sequence.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this function.

zl:length *x* — Counts the elements in the list *x*. Returns a non-negative integer. Use Common Lisp function, **length**.

Predicates that Operate on Sequences

The predicates take as many arguments as there are sequences provided. The argument *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 sequence is reached.

some *predicate &rest sequences*
: Each element in *sequences* 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 *sequences* 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 *sequences* 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 *sequences* is tested against *predicate*. Returns non-**nil** as soon as any invocation of *predicate* returns **nil**. Otherwise returns **nil**.

Note: The following Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalent of these predicates.

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 *pred* returns non-**nil** when applied to, or **nil** if *pred* returns **nil** for every element.

zl:every *list pred &optional (step #'cdr)*
: Each element, default *step*, in *list* is tested against *pred*. Returns **t** if *pred* returns non-**nil** when applied to every element of *list*, or **nil** if *pred* returns **nil** for some element.

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. The function **map** can be applied to any kind of sequence, but the other map-type functions operate only on lists. The function **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*.

map-into *result-sequence function* &rest *sequences*

Destructively modifies the *result-sequence* to contain the results of applying the *function* to each element in the argument *sequences* in turn.

reduce *function sequence &key from-end* (*start 0*) *end* (*initial-value nil initial-value-p*)

Combines the elements of *sequence*, using a binary operation. Returns the result of using *function* on *sequence*.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this function.

zl:map *fcn list &rest more-lists*

Applies *fcn* to *list* and to successive sublists of that list. Returns a new list, such that sublist *i* of the new list is the result of applying *function* to sublist *i* of each of *more-lists*. Use the Common Lisp function **mapl**.

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**.

reverse *sequence*

Returns a new sequence of the same type as *sequence*,containing the same elements in reverse order.

nreverse *sequence* — Returns a sequence containing the same elements as *sequence*, but in reverse order. This is a destructive version of **reverse**.

fill *sequence item* &key *(:start* **0***) :end*

Destructively modifies *sequence* by replacing each element of the subsequence specified by the **:start** (which defaults to zero) and **:end** (which defaults to the length of the sequence) arguments with *item*.

replace *sequence1 sequence2* &key *(:start1* **0***) :end1 (:start2* **0***) :end2*

Destructively modifies *sequence1* by copying into it successive elements from *sequence2*.

remove-duplicates *sequence* &key *:from-end (:test* **#'eql***) :test-not (:start* **0***) :end :key*

Compares the elements of *sequence* pairwise, and if any two match, then the one occurring earlier in the sequence is discarded.

delete-duplicates *sequence* &key *(:test* **#'eql***) :test-not (:start* **0***) :end :from-end :key :replace*

Compares the elements of *sequence* pairwise, and if any two match, then the one occurring earlier in the sequence is discarded. This is a destructive function.

substitute *newitem olditem sequence* &key *(:test* **#'eql***) :test-not (:key* **#'identity***) :from-end (:start* **0***) :end :count*

Returns a sequence of the same type as *sequence* that has the same elements, except that those in the subsequence delimited by **:start** and **:end** and satisfying the predicate specified by the **:test** keyword are replaced by *newitem*.

substitute-if *newitem predicate sequence* &key *:key :from-end (:start* **0***) :end :count*

Returns a sequence of the same type as *sequence* that has the same elements, except that those in the subsequence delimited by **:start** and **:end** and satisfying *predicate* arereplaced by *newitem*.

substitute-if-not *newitem predicate sequence* &key *:key :from-end (:start* **0***) :end :count*

Returns a sequence of the same type as *sequence* that has the same elements, except that those in the subsequence delimited by **:start** and **:end** that do not satisfy *predicate* are replaced by *newitem*.

nsubstitute *newitem olditem sequence* &key *(:test* **#'eql***) :test-not (:key* **#'identity***) :from-end (:start* **0***) :end :count*

Returns a sequence of the same type as the argument *sequence* which has the same elements, except that those in the subsequence delimited by **:start** and **:end** and satisfying the predicate specified by the **:test** keyword have been replaced by *newitem*. This is a destructive version of **substitute**.

nsubstitute-if *newitem predicate sequence* &key *:key :from-end (:start* **0***) :end :count*
Returns a sequence of the same type as the argument *sequence* which has the same elements, except that those in the subsequence delimited by **:start** and **:end** and satisfying *predicate* have been replaced by *newitem*. This is a destructive version of **nsubstitute**.

nsubstitute-if-not *newitem predicate sequence* &key *:key :from-end (:start* **0***) :end :count*
Returns a sequence of the same type as the argument *sequence* which has the same elements, except that those in the subsequence delimited by **:start** and **:end** which do not satisfy *predicate* have been replaced by *newitem*. This is a destructive version of **substitute-if-not**.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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. This is a destructive version of **zl:reverse**.

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 *sequence*. Returns the new sequence.

remove-if *predicate sequence* &key *:key :from-end* (*:start* **0**) *:end :count*
Non-destructively removes those items from the *sequence* that satisfy *predicate*. Returns the new sequence.

remove-if-not *predicate sequence* &key *:key :from-end* (*:start* **0**) *:end :count*
Non-destructively removes those items from *se-*

quence that do not satisfy *predicate*. Returns the new sequence.

remove-duplicates *sequence* &key *:from-end* (*:test* **#'eql**) *:test-not* (*:start* **0**) *:end* *:key*

Non-destructively removes duplicate elements from *sequence*. Returns the new sequence.

delete *item sequence* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) *:from-end* (*:start* **0**) *:end* *:count*

Destructive version of **remove**. Returns the modified *sequence*.

delete-if *predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end* *:count*

Destructive version of **remove-if**. Returns the modified *sequence*.

delete-if-not *predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end* *:count*

Destructive version of **remove-if-not**. Returns the modified *sequence*.

delete-duplicates *sequence* &key (*:test* **#'eql**) *:test-not* (*:start* **0**) *:end* *:from-end* *:key* *:replace*

Destructive version of **remove-duplicates**. Returns the modified *sequence*.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:remove *item list* &optional (*times* **most-positive-fixnum**)

Non-destructive version of **zl:delete**. Use the Common Lisp function **remove**.

zl:delete *item list* &optional (*ntimes* **-1**)

Deletes occurrences of *item* from *list* (**equal** is used for the comparison). Returns the *list* with all occurrences of *item* removed. Use the Common Lisp function **delete**.

Replacing Sequences

fill *sequence item* &key (*:start* **0**) *:end*

Destructively replaces each element of *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 *olditem* for *newitem* in *sequence*. Returns the new sequence.

substitute-if *newitem predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end* *:count*
Non-destructively replaces elements in *sequence* that satisfy *predicate* with *newitem*. Returns the new sequence.

substitute-if-not *newitem predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end* *:count*
Non-destructively replaces elements in *sequence* that do not satisfy *predicate* with *newitem*. Returns the new sequence.

nsubstitute *newitem olditem sequence* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) *:from-end* (*:start* **0**) *:end* *:count*
Destructive version of **substitute**. Returns the modified *sequence*.

nsubstitute-if *newitem predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end* *:count*
Destructive version of **substitute-if**. Returns the modified *sequence*.

nsubstitute-if-not *newitem predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end* *:count*
Destructive version of **substitute-if-not**. Returns the modified *sequence*.

Searching for Sequence Items

Each of the searching functions searches a sequence 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 *sequence*. Returns *item* if found, otherwise **nil**.

find-if *predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end*
Finds the leftmost element in *sequence* that satisfies *predicate*. Returns said element if found, otherwise **nil**.

find-if-not *predicate sequence* &key *:key* *:from-end* (*:start* **0**) *:end*
Finds the leftmost element in *sequence* that does not satisfy *predicate*. Returns said element if found, otherwise **nil**.

position *item sequence* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) *:from-end* (*:start* **0**) *:end*

Finds the leftmost *item* in *sequence*. Returns the index of the item if found, otherwise **nil**.

position-if *predicate sequence* &key *:key :from-end* (*:start* **0**) *:end*

Finds the leftmost element in *sequence* that *predicate*. Returns the index of the element if found, otherwise **nil**.

position-if-not *predicate sequence* &key *:key :from-end* (*:start* **0**) *:end*

Finds the leftmost element in *sequence* that does satisfy *predicate*. Returns the index of the element if found, otherwise **nil**.

count *item sequence* &key (*:test* **#'eql**) *:test-not* (*:key* **#'identity**) *:from-end* (*:start* **0**) *:end*

Counts the elements in the specified subsequence of *sequence* that satisfy predicate specified by **:test**. Returns the result.

count-if *predicate sequence* &key *:key :from-end* (*:start* **0**) *:end*

Counts the elements in the specified subsequence of *sequence* that satisfy *predicate*. Returns the result.

count-if-not *predicate sequence* &key *:key :from-end* (*:start* **0**) *:end*

Counts elements in the specified subsequence of *sequence* that do not satisfy *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.

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 *sequence1* and *sequence2* according to an order determined by *predicate*. Returns merged sequences.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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 Common Lisp function **sort**.

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 Common Lisp function **stable-sort**.

Sequence Functions in the CL Package with SCL Extensions

Here are the sequence functions that have Symbolics Common Lisp extensions:

Function	*Extension(s)*
copy-seq	*area*
delete-duplicates	*:replace*
make-sequence	*:area*

Characters

For an introduction to characters, see the section "Overview of Characters".

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)

For more information on bit keys, see the section "Using Modifier Keys".

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

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 #\Space #\Newline

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".

For a list of character type-checking predicates, see the section "Character Predicates".

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.

Fields of a Character

The following diagram depicts the fields of a character:

```
|<-----------------Entire character---------------->|
|<---Bits--->|<--Style--->|<-------Char code------->|
                          |<-Char set->|<-Subindex->|
```

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.

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".

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 ! shows the characters in the mouse character set. Figure ! 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

--mouse--

Up-Arrow
Right-Arrow
Down-Arrow
Left-Arrow
Vertical-Double-Arrow
Horizontal-Double-Arrow
NW-Arrow
Times
Fat-Up-Arrow
Fat-Right-Arrow
Fat-Down-Arrow
Fat-Left-Arrow
Fat-Double-Vertical-Arrow
Fat-Double-Horizontal-Arrow
Paragraph
NW-Corner
SE-Corner
Hourglass
Circle-Plus
Paintbrush
Scissors
Trident
NE-Arrow
Circle-Times
Big-Triangle
Medium-Triangle
Small-Triangle
Inverse-Up-Arrow
Inverse-Down-Arrow
Plus
Filled-Lozenge
Hollow-UP-Arrow
Hollow-NW-Arrow
Hollow-NE-Arrow
Dot
Fat-Times
Small-Filled-Circle
Filled-Circle
Fat-Circle
Fat-Circle-Minus
Fat-Circle-Plus
Down-Arrow-To-Bar
Short-Down-Arrow
Up-Arrow-To-Bar
Short-Up-Arrow
Boxed-Up-Triangle
Boxed-Down-Triangle
Fat-Plus
Maltese-Cross

Figure 14. Mouse Character Set

Character Code, Bits, and Style

--arrow--

Center-Dot	Left-Arrowhead-Dot	Right-Arrowhead
Circle-Plus ⊕	Right-Triangle ▶	Right-Open-Arrow ⇒
Circle-Cross ⊗	Up-Open-Arrow ⇑	Baseline-Caret ▴
Down-Arrowhead ˅	Right-Hand ☞	Right-Short-Open-Arrow ⇒
Up-Arrowhead ˄	Left-Hand ☜	Open-X ⌘
Right-Fat-Arrow ➽	Eye	
Right-Arrowhead-Dot	Left-Arrowhead <	

Figure 15. Arrow Character Set

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".

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 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.

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.

Note that **eq** might not work for characters in other implementations of the Common Lisp dialect.

Character Styles

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.

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 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".

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
SWISS.BOLD.SAME	FIX.ROMAN.NORMAL	SWISS.BOLD.NORMAL

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.

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, see the section "Character Styles for TV Fonts". For a mapping of styles for the LGP2/LGP3 printer, see the section "Character Styles for LGP2/LGP3 Fonts".

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/LGP3 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 ! shows how the characters of the five most common sizes appear for a given family and face. The faces of the displayed characters from left to right are: NORMAL, ITALIC, BOLD, BOLD-ITALIC.

Figure ! shows how the characters of the four most common faces appear for a given family and size. The families of the displayed characters from left to right are: FIX, SWISS, DUTCH, and JESS.

```
      | 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 16. Varying Character Sizes: VERY-SMALL to VERY-LARGE

```
           | 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 17. Varying Character Faces: ROMAN, ITALIC, BOLD, BOLD-ITALIC

Figure ! shows how the characters of the four most common families appear for a given face and size. The sizes of the displayed characters from left to right are: very-small, small, normal, large, and very-large.

```
      | 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 18. Varying Character Families: FIX, SWISS, DUTCH, JESS

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:

See the section "Using Character Styles in the Input Editor".
See the section "Character Styles and the Lisp Listener".
See the section "Using Character Styles in Zmail".
See the section "Using Character Styles in Hardcopy".
See the section "Using Character Styles in Zmacs".

Refer to the following sections for descriptions of facilities for programming with character styles:

See the function **with-character-style**.
See the function **with-character-family**.
See the function **with-character-face**.
See the function **with-character-size**.

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.

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*, 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.

A *font object* may be various things, depending on the device.

If *error-p* is non-**nil**, this function signals an error if no mapping to a font is found. If *error-p* is **nil** and no font mapping is found, **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"
```

For related information: See the section "Mapping a Character Style to a Font".

si:define-character-style-families *device character-set* &rest *plists* *Function*

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 or LGP3 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/LGP3 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".

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**".

For related information: See the section "Mapping a Character Style to a Font".

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"))))
```

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-style-families** 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 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/LGP3 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-FONT 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**.

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
```

Character Styles for LGP2/LGP3 Fonts

Family FIX: Fonts are supported for all combinations of faces ROMAN, ITALIC, BOLD, and BOLD-ITALIC and sizes TINY, VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE.

Face	*LGP2/LGP3 Font*
ROMAN	Courier
ITALIC	Courier-Oblique
BOLD	Courier-Bold
BOLD-ITALIC	Courier-Bold-Oblique

Size	*LGP2/LGP3 Font Size*
TINY	4 point
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, ITAL-IC, BOLD, and BOLD-ITALIC and sizes VERY-SMALL, SMALL, NORMAL, LARGE, VERY-LARGE.

Face	*LGP2/LGP3 Font*
ROMAN	Helvetica
ITALIC	Helvetica-Oblique
BOLD	Helvetica-Bold
BOLD-ITALIC	Helvetica-Bold-Oblique

Size	*LGP2/LGP3 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/LGP3 Font*
ROMAN	Times-Roman
ITALIC	Times-Oblique
BOLD	Times-Bold
BOLD-ITALIC	Times-Bold-Oblique

Size	*LGP2/LGP3 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/LGP3 Font*
ROMAN	Times-Roman
ITALIC	Times-Oblique
BOLD	Times-Bold
BOLD-ITALIC	Times-Bold-Oblique

Size	*LGP2LGP3 Font Size*

VERY-SMALL	8 point
SMALL	9 point
NORMAL	12 point
LARGE	15 point
VERY-LARGE	19 point

Tables of Character Functions

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.

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.

Character Fields

For a diagram of the fields of a character, see the section "Fields of a Character".

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.

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".

Character Comparisons

None of the character comparisons ignore the character's bits.

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.

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.

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.

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.

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.

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.

The Character Set

Characters in the Symbolics standard character set whose codes are less than 200 octal (with the 200 bit off), and only those, are "printing graphics"; when output to a device they are assumed to print a character and move the "cursor" one character position to the right. (All software provides for variable-width character styles, so the term "character position" should not be taken too literally.)

Characters in the range of 200 to 236 inclusive are used for special characters. Character 200 is a "null character", which does not correspond to any key on the keyboard. The null character is not used for anything much. Characters 201 through 236 correspond to the special function keys on the keyboard such as RETURN. Some characters are reserved for future expansion.

It should never be necessary for a user or a source program to know these numerical values. Indeed, they are likely to be changed in the future. There are symbolic names for all characters; see below.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Symbolics characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding, and the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, will not be the same measured in original Symbolics characters as it will measured in the encoded ASCII characters.

In TOPS-20, Tenex, and ITS, in the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (delete/vt), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a "quoting character" for these codes. Codes 210 (overstrike), 211 (tab), 212 (line), and 214 (page), are converted to their ASCII cognates,

namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

Most of the special characters do not normally appear in files (although it is not forbidden for files to contain them). These characters exist mainly to be used as "commands" from the keyboard.

A few special characters, however, are "format effectors" which are just as legitimate as printing characters in text files. The following is a list of the names and meanings of these characters:

Return	The "carriage return" character which separates lines of text. Note that the PDP-10 convention that lines are ended by a pair of characters, "carriage return" and "line feed", is not used.
Page	The "page separator" character which separates pages of text.
Tab	The "tabulation" character which spaces to the right until the next "tab stop". Tab stops are normally every 8 character positions.

The Space character is considered to be a printing character whose printed image happens to be blank, rather than a format effector.

There are some characters which are not typeable as keys on a Symbolics 3600 console, even though there are codes and names for such characters. Those characters are:

205 Macro	220 Stop-Output	231 Hand-Up
216 Quote	223 Status	233 Hand-Left
217 Hold-Output	230 Roman-IV	234 Hand-Right

The Symbolics standard character set consists of mappings for the octal codes 000-241. The codes 242-377 are unused in this character set. The names of the characters are in the table in sys:io;rddefs.lisp. Figure ! is a table of the code mappings.

Strings

Introduction to Strings

Strings are a type of one-dimensional array (a vector) whose elements are characters.

Symbolics Common Lisp supports two types of strings whose elements are of 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 strings are also referred to as *thin* strings and *fat* strings.

000 · Center-Dot	040 Space	100 @	140 ‘
001 ↓ Down-Arrow	041 !	101 A	141 a
002 α Alpha	042 "	102 B	142 b
003 β Beta	043 #	103 C	143 c
004 ∧ And-sign	044 $	104 D	144 d
005 ¬ Not-sign	045 %	105 E	145 e
006 ε Epsilon	046 &	106 F	146 f
007 π Pi	047 ’	107 G	147 g
010 λ Lambda	050 (Open	110 H	150 h
011 γ Gamma	051) Close	111 I	151 i
012 δ Delta	052 *	112 J	152 j
013 ↑ Up-Arrow	053 + Plus-sign	113 K	153 k
014 ± Plus-Minus	054 ,	114 L	154 l
015 ⊕ Circle-Plus	055 - Minus-sign	115 M	155 m
016 ∞ Infinity	056 .	116 N	156 n
017 ∂ Partial-Delta	057 /	117 O	157 o
020 ⊂ Left-Horseshoe	060 0	120 P	160 p
021 ⊃ Right-Horseshoe	061 1	121 Q	161 q
022 ∩ Up-Horseshoe	062 2	122 R	162 r
023 ∪ Down-Horseshoe	063 3	123 S	163 s
024 ∀ Universal-Quantifier	064 4	124 T	164 t
025 ∃ Existential-Quantifier	065 5	125 U	165 u
026 ⊗ Circle-X	066 6	126 V	166 v
027 ↔ Double-Arrow	067 7	127 W	167 w
030 ← Left-Arrow	070 8	130 X	170 x
031 → Right-Arrow	071 9	131 Y	171 y
032 ≠ Not-Equals	072 :	132 Z	172 z
033 ◊ Lozenge	073 ;	133 [	173 {
034 ≤ Less-Or-Equal	074 < Less-sign	134 \	174 |
035 ≥ Greater-Or-Equal	075 = Equal-sign	135]	175 }
036 ≡ Equivalence	076 > Greater-sign	136 ^	176 ~
037 ∨ Or-sign	077 ?	137 _	177 ∫ Integral
200 Null	210 Back-Space	220 Stop-Output	230 Roman-IV
201 Suspend	211 Tab	221 Abort	231 Hand-Up
202 Clear-Input	212 Line	222 Resume	232 Scroll
203 Reserved	213 Refresh	223 Status	233 Hand-Left
204 Function	214 Page	224 End	234 Hand-Right
205 Macro	215 Return	225 Square	235 Select
206 Help	216 Quote	226 Circle	236 Network
207 Rubout	217 Hold-Output	227 Triangle	237 Escape
240 Complete			
241 Symbol-Help			

Figure 19. The Symbolics Standard Character Set

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. A fat string can hold any character, including characters which are too large to be elements of a thin string.

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
```

See the section "The Character Set". The way characters work, including multiple fonts and the extra bits from the keyboard, is explained in that section.

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". See the section "Displaced Arrays". Fill-pointers and the concept of shared arrays are discussed in those 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) => "DDD"
(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) => "DD"
(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 a 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". 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.

Strings can be used to create symbols. The function **intern**, for example, given a string, returns "the" symbol with that print name. Similarly, **make-symbol** creates and returns an uninterned symbol with that print name.

How the Reader Recognizes Strings

The reader recognizes strings, written as a sequence of the characters contained in the string, and enclosed in double quotes (" ").

Any double quote or escape character, for example, the \ (backslash), in the sequence must be preceded by another \ escape character.

Zetalisp 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 **prin1** 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.

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 ! 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".

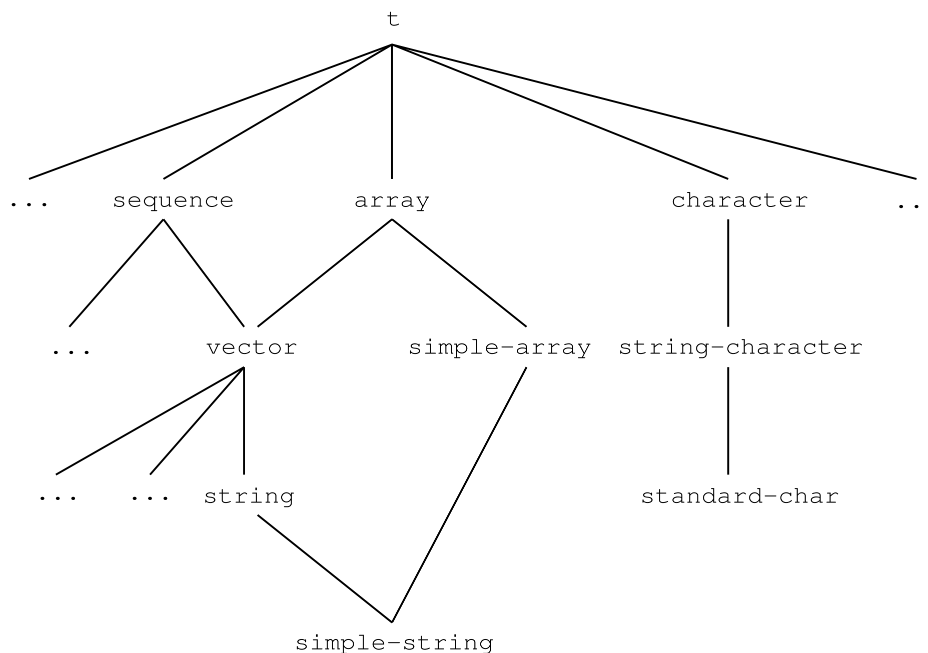


Figure 20. Symbolics Common Lisp String Data Types

A string is a specialized vector, or one-dimensional array, whose elements are of type **character**, or **string-char**.

In Lisp terms: `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".

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".

Operations with Strings

Several types of string operations 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 names begin 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 nine major groups, as follows:

- Type-checking Predicates
- Access and Information
- Construction
- Conversion (case changes and pluralizing)
- Manipulation (trimming and reversing)
- Comparison Predicates (case-sensitive and case-insensitive comparisons)
- Searches (case-sensitive and case-insensitive searches)
- ASCII Conversion
- Input and Output

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".

Several string functions, notably those involving searches and comparison, are further subdivided into groups that either respect or ignore string characteristics such as character style and alphabetic case.

See the section "Case-Sensitive and Case-Insensitive String Comparisons".

When strings have fill-pointers, string functions generally operate only on the active portion of the string (beyond 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".

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.

Case-Sensitive and Case-Insensitive String Comparisons

String comparisons compare every individual element of the string arrays 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". Here, for convenience, is a summary of some pertinent character attribute information:

Character style — A combination of three characteristics that describe how a character appears: 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 (small) and upper (capital) case for alphabetic characters.

Character code — Identifies the character within its character set, in the same way that ASCII codes represent particular characters.

Modifier bits Refer 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 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
A        a        char= NIL char-equal T
A        c-A      char= NIL char-equal NIL
A        A        char= NIL char-equal T
a        a        char= T   char-equal T
a        c-A      char= NIL char-equal NIL
a        A        char= NIL char-equal T
c-A      c-A      char= T   char-equal T
c-A      A        char= NIL char-equal NIL
A        A        char= T   char-equal T
NIL
```

Keyword Arguments Delimit and Direct String Operations

For the sake of efficiency, the majority of string-specific functions let you operate on a portion of a string. 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 beyond the limit specified by the fill pointer, if there is one.

:start must be smaller than, or equal to, **:end**, otherwise 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 you omit both **:start** and **:end**, the entire string is processed by default.

If two strings are involved, you can use the keyword arguments **:start1**, **:end1**, **:start2**, and **:end2** 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. You can reverse the conceptual direction of a search by using the keyword argument **:from-end**.

Search functions normally process strings in the forward direction, but if you specify a non-**nil** value for **:from-end**, processing starts from the reverse direction. See the section "Case-Insensitive String Searches".

For some functions, you can specify how many occurrences of an item should be affected with the keyword **:count**. If **:count** is **nil**, or is not supplied, all matching items are affected.

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".

simple-string-p *object* — Determines if *object* is a simple string array (one with no fill pointer and no displacement), returning **t** if it is, and **nil** otherwise. Accepts any object as an argument.

string-char-p *char* — Determines if *char* can be stored into a thin string (that is, if it is a standard character), returning **t** if it can, and **nil** otherwise. Accepts a character argument only.

string-fat-p *string* — Determines if *string* is an array of fat characters, returning **t** if it can, and **nil** otherwise. Accepts a string argument only.

stringp *object* — Determines if if *object* is either type of string, returning **t** if it is, and **nil** otherwise. Accepts any object as an argument.

String Access and Information

This group includes functions that access a string either to extract a one or more characters (a substring), 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**, since in many cases **char** is more efficient than **aref**. Another advantage of using **char** is that readers of your code can tell that you are working with a string.

You can use the function **setf** 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**.

string-a-or-an *string* &optional *(both-words* **t**) *(case* **:downcase**)
Computes whether the article "a" or "an" is used when introducing a noun. If *both-words* is true, the result is the concatenation of the article, a space, and the *noun*; otherwise, the article is returned.

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*.

sys:number-into-array *array n* &optional (*radix* **zl:base**) (*at-index* 0) (*min-columns* 0)
Deposits the printed representation of *number* into *array*.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, if possible, use the Common Lisp equivalent this function.

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 the Common Lisp **parse-integer**.

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 **prin1-to-string**, **princ-to-string**, or **format**.

string *x*

Coerces *x* into a string. Returns a string if *x* is a string, or 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*

Creates 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*

Concatenates copies of its string arguments into a single string and returns that 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.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this function.

zl:string-nconc *to-string* &rest *strings*

Returns a concatenated string but modifies its first argument instead of copying it. Use the Common Lisp function **string-nconc**.

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 beginning of the 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 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**.

string-pluralize *string*
Creates and returns a string that is the plural of *string*.

string-upcase *string* &key *(start 0) (end* **nil**)
Creates and returns a copy of string with all lowercase characters capitalized.

nstring-upcase *string* &key (*start* 0) (*end* **nil**)
Returns *string* with all lowercase characters capitalized.

string-downcase *string* &key *(start 0) (end* **nil**)
Creates and returns a copy of string with all uppercase characters replaced by lowercase.

nstring-downcase *string* &key (*start* 0) (*end* **nil**)
Returns *string* with all uppercase characters replaced by lowercase.

string-capitalize *string* &key *(start 0) (end* **nil**)
Creates and 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**)
Creates and 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 and with hyphens changed to spaces.

string-flipcase *string* &key *(start 0) (end* **nil**)
Creates and 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.

The Zetalisp versions of these functions have optional starting and ending arguments. A further Zetalisp 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).

Note: These Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:string-pluralize *string*
Returns a string containing the plural of the word in *string*. Use the Common Lisp **string-pluralize**.

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 the Common Lisp functions **string-upcase** and **nstring-upcase**.

zl:string-downcase *string* &optional (*start* **0**) *to* (*copy-p* **t**)
Returns a copy of *string* with uppercase characters replaced by lowercase or *string* itself is modified and returned. Use the Common Lisp functions **string-downcase** and **nstring-downcase**.

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 the Common Lisp functions **string-capitalize-words** and **nstring-capitalize-words**.

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 the Common Lisp functions **string-flipcase** and **nstring-flipcase**.

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 from a specified location within the string or from either extremity.

Some string manipulation functions take the argument *char-set*. This argument can be 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".

string-trim *char-set string*
: Strips the characters *char-set* off the beginning and end of *string*, and returns the resulting substring.

string-thin *string* &key (*:start* 0) *:end* (*:remove-style* **t**) *:remove-bits :error-if :area*
: Strips the specified character-style information and bucky bits from *string*, and returns the resulting substring. (Hyper, meta, super, and control are bits.) *String* is an array of characters.

string-left-trim *char-set string*
: Strips the characters in *char-set* of the beginning of *string*. Returns a substring of *string*.

string-right-trim *char-set string*
: Strips the characters in *char-set* off the end of *string*. Returns a substring of *string*.

string-reverse *string* &key (*start* 0) (*end* **nil**)
: Creates and 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**.

In Zetalisp, *char-set* can be a list of characters, array of characters, or a string of characters.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:string-trim *char-set string*
: Strips the characters in *char-set* off the beginning and end of *string*, and returns the resulting substring. Use the Common Lisp function **string-trim**.

zl:string-left-trim *char-set string*
: Strips the characters in *char-set* off the beginning of *string*. Returns a substring of *string*. Use the Common Lisp function **string-left-trim**.

zl:string-right-trim *char-set string*
: Strips the characters in *char-set* from the end of *string*. Returns a substring of *string*. Use the Common Lisp function **string-right-trim**.

zl:string-reverse *string*
: Creates and returns a copy of *string* with the order of characters reversed. Use the Common Lisp function **string-reverse**.

zl:string-nreverse *string*
: Returns *string* with the order of characters reversed. Use the Common Lisp function **string-nreverse**.

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".

The keywords *:start1* **0** and *:start2* **0** 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 **nil**. 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 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.

string= *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if two strings are are identical in all character fields, including modifier bits, character set, character style, and alphabetic case; it is false otherwise.

string≠ *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if the characters in the two strings are not identical (same as **user::string////=**).

user::string////= *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
A synonym of **string≠**.

string< *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
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* (*:start2* **0**) *:end2*
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* (*:start2* **0**) *:end2*
A synonym of **string≤**.

string> *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
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* (*:start2* **0**) *:end2*
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* (*:start2* **0**) *:end2*
A synonym of **string≥**.

string-exact-compare *string1 string2* &key (*:start1* **0**) (*:start2* **0**) *:end1 :end2*

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*.

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.

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**.

Note: These Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:string-exact-compare *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2*

Returns a positive number is *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. Use the Common Lisp function **string-exact-compare**.

zl:string= *string1 string2* &optional *idx1 idx2 lim1 lim2*

Like **string=**, but returns **t** or **nil**.

zl:string≠ *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2*

Like **string≠**, but returns **t** or **nil**.

zl:string< *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *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**.

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".

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** (the beginning of the string); the end arguments default to **nil** (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 of the first character at which the strings fail to match; this index is equivalent to the length of the common portion of the strings.

These predicates ignore the character fields for character style and alphabetic case for the comparison.

string-equal *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Tests if two strings are are identical in all character fields, including modifier bits, character set, and character style; otherwise it is false. Case-insensitive version of **string=**.

string-not-equal *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Test if *string1* is not equal to *string2*. If the condition is satisfied, **string-not-equal** returns the position within the strings of the first character at which the strings fail to match. Case-insensitive version of **user::string////=**.

string-lessp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Tests if the first characters that differ between *string1* and *string2* are **char<**, or if *string1* is a proper substring of *string2*. Case-insensitive version of **string<**.

string-greaterp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Tests if the first characters that differ between *string1* and *string2* are **char>**, or if *string2* is a proper substring of *string1*. Case-insensitive version of **string>**.

string-not-greaterp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Tests if *string1* is less than or equal to *string2*. If the condition is satisfied, **string-not-greaterp** returns the position within the strings of the first character at which the strings fail to match. Case-insensitive version of **string<=**.

string-not-lessp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Tests if *string1* is greater than or equal to *string2*. If the condition is satisfied, **string-not-lessp** returns the position within the strings of the first character at which the strings fail to match. Case-insensitive version of **string>=**.

string-compare *string1 string2* &key (*:start1* **0**) (*:start2* **0**) *:end1 :end2*

Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. 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. Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. Case-insensitive version of **sys:%string-exact-compare**.

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**.

These predicates ignore the character fields for character style and alphabetic case for the comparison.

Note: These Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:string-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings, returning **t** if they are equal and **nil** if they are not. Case-insensitive version of **zl:string=**. Use the Common Lisp function **string-equal**.

zl:string-not-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings or substrings of them. 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*

Compares two strings using alphabetical order. 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>**. This compares two strings or substrings of them. Like **string-greaterp** but returns **t** or **nil**.

zl:string-not-lessp *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings, or substrings of them. Like **string-not-lessp** but returns **t** or **nil**.

zl:string-not-greaterp *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings or substrings of them. Like **string-not-greaterp** but returns **t** or **nil**.

zl:string-compare *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares the characters of *string1* starting at *idx1* and ending just below *lim1* with the characters of *string2* starting at *idx2* and ending just below *lim2*. Case-insensitive version of **zl:string-exact-compare**. Use the Comon Lisp function **string-compare**.

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". 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** (the beginning of the string); the end arguments default to **nil** (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. 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".

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".

string-search-exact-char *char string* &key *:from-end* (*:start* **0**) *:end*
: 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*
: 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* (*:start2* **0**) *:end2*
: 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*.

sys:%string-search-exact-char *char string start end*
: A low-level search, possibly more efficient than other searching functions.

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 (the beginning of the string), while *to* defaults to **nil** (the length of the string). *key-start* and *key-end* default in analogous fashion.

Zetalisp has separate reverse-search functions. These return a string position counted from the *beginning* of the string, even though the search begins at the *end* of the string.

Note: These Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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 Common Lisp function **string-search-not-exact-char**.

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 Common Lisp function **string-search-exact**.

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*.

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 alpha-

betic case. See the section "Case-Sensitive and Case-Insensitive String Comparisons".

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** (the beginning of the string); 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*, which can be 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 given with 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".

General sequence functions can also be used to locate one or more string elements satisfying some test. See the section "Searching for Sequence Items".

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**.

string-search-char *char string* &key *:from-end* (*:start* **0**) *:end*
: 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*
: 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* (*:start2* **0**) *:end2*
: 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*
: 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 0) *char-set string* &key *:from-end* (*:start* **0**) *:end*
: 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*.

sys:%string-search-char *char string start end*
A low-level, possibly more efficient search.

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 (the beginning of the string), while *to* defaults to **nil** (the length of the string). *key-start* and *key-end* default in analogous fashion.

The Zetalisp argument *char-set* can be represented in Zetalisp as a list of characters or a string of characters.

Zetalisp has 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.

Note: These Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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 Common Lisp function **string-search-char**.

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 Common Lisp function **string-search-not-char**.

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 Common Lisp function **string-search**.

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*.

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 Common Lisp function **string-search-set**.

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 Common Lisp function **string-search-not-set**.

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*.

Summary of String Searching Functions

Case-sensitive Version	*Case-insensitive Version*
string-search-exact-char	**string-search-char**
string-search-not-exact-char	**string-search-not-char**
string-search-exact	**string-search**
sys:%string-search-exact-char	**sys:%string-search-char**
[None]	**string-search-set**
[None]	**string-search-not-set**

ASCII Conversion 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".

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.

String input and output functions, and the two variables that control the printing of strings, are summarized elsewhere. See the section "String Input and Output Functions". See the section "Control Variable for Printing Strings".

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".

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 function **with-open-file**.

with-output-to-string *(stream* &optional *string* &key *:index)* &body *body*

Executes *body* with *stream* bound to a character output stream; all output to that stream is saved in a string. See the function **with-open-file**.

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".

make-string-output-stream

Returns an output stream that accumulates output for **get-output-stream-string**. See the section "Stream Operations".

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".

write-to-string *object* &key *:escape :radix :base :circle :pretty :level :length :case :gensym :array :integer-length :array-length :string-length :bit-vector-length :abbreviate-quote :readably :structure-contents :exact-float-value*

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".

read-from-string *string* &optional (*eof-errorp* **t**) *eof-value* &key (*start* **0**) *end preserve-whitespace*

Gives the characters of *string* successively to the Lisp reader, and returns the Lisp object built by the reader. Returns the object read, and the index of the first character in the string not read (if the entire string was read, returns the length of the string). See the section "Input Functions".

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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 function **with-open-file**.

zl:with-output-to-string *(var* &optional *string index)* &body *body*

Provides a variety of ways to send output to a string through an I/O stream. See the function **with-open-file**.

zl:read-from-string *string* &optional (*eof-option* **'si:no-eof-option**) (*start* **0**) *end* (*preserve-whitespace* **zl:read-preserve-delimiters**)

Gives the characters of *string* successively to the reader, and returns the Lisp object built by the reader. See the section "Input Functions".

Control Variable for Printing Strings

print-string-length Controls the number of string characters to print. See the section "Output Functions".

Table Management

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 that 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*. More complex tables can have 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 a bird of that type 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.

Genera's 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, your code is easily ported to Common Lisp, and you take advantage of the efficiencies provided by the facility. If you create tables that do not use the Symbolics extensions to the **make-hash-table** function, your code is already compatible with Common Lisp.

Note: In figuring out the best internal representation for the given data, the table management facility uses a small amount of overhead. Thus, if you know beforehand that you need a simple table, for instance a property list or an association list, it may be more efficient to create your own list rather than use the table management facility to do it.

Using this scheme for both simple and complex tables has advantages.

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 the facility offers there are other advantages. It is easy, for example, to extend this facility to create tables with your own mixins for customization.

In addition, 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 which make them highly efficient.

The Zetalisp function **zl:make-hash-table** is obsolete. 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, old-style tables can be more useful at times. These features are covered elsewhere. See the section "Other Data Types Used as Tables".

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**, and **hash-table-count**. Genera hash tables do not necessarily hash, they only use hashing when the table requires it.

Hash Table Interface to the Table Management Facility

For the most common types of tables, a table object 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. A table object may have zero values associated with it, if the value of **:number-of-values** is 0.

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 **setf** macro with the **gethash** function. To look up a key and find the associated value, use the **gethash** function.

Note: 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". Here is an example of how to create and work with a table that describes a plant:

```
(setq plant (make-hash-table :size 10))
   => #<Table 0/0 71065203>

(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)
   => #<Table 4/4 71065203> is a table with 4 entires.
        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
   #<Table 4/4 71065203>
```

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 prints 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. 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.

Table Functions

Here is a list of functions that operate specifically on tables:

clrhash *table*
: Returns *table* after removing all of its entries.

gethash *key table* &optional *default*
: Finds the value associated with a particular key. Returns three values: the value; **t** or **nil** depending on whether key was found; and *key* if it was found. (This is the *key* with which the table was originally created.)

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
: &rest *options* &key (*:test* **'eql**) (*:size* **cli:*default-table-size***) (*:area* **default-cons-area**) *:hash-function :rehash-before-cold :rehash-after-full-gc* (*:number-of-values* **1**) *:store-hash-code* (*:mutating* **t**) *:initial-contents :ignore-gc :growth-factor :growth-threshold :rehash-size :rehash-threshold*
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*
: 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*. (The *key* is the one with which the table was originally created.)

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 an entry for *key* already exists, it replaces the value of that entry with *value*. **swaphash** returns two values: the old value

and a flag. If there is an old value, the flag is **t**.

When a table object is created, it has a *size*, which is the number of entries it can hold. This number, however, is simply 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 association list, and it grows past an efficient size for association lists, 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 objects 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**.

Creating Table Objects

You use **make-hash-table** to make a new table.

Many initialization options to **make-hash-table** are available for customizing a table to your application.

make-hash-table &key *:name (:test* **'eql***) (:size* **cli:*default-table-size****) (:area* **sys:default-cons-area***) :hash-function :rehash-before-cold :rehash-after-full-gc (:number-of-values* **1***) :store-hash-code (:gc-protect-values* **t***) (:mutating* **t***) :initial-contents :optimizations (:locking* **:process***) :ignore-gc (:growth-factor* **cli::*default-table-growth-factor****) (:growth-threshold* **cli::*default-table-growth-threshold****) :rehash-size :rehash-threshold* *Function*

Creates and returns a new table object. This function calls **make-instance** using a basic table flavor and mixins for the necessary additional flavors as specified by the options.

make-hash-table takes the following keyword arguments:

:name A symbol that identifies the table in progress notes.

:test Determines how keys are compared. Its argument can be any function; **eql** is the default. If you supply one of the following values or predicates the hash table facility automatically supplies a **:hash-function**: **eq**, **eql**, **equal**, **char-equal**, **char=**, **string-equal**, **#'string-equal**, **string=**, **zl:equal**, **zl:string-equal**, **zl:string=**. If you supply a value or predicate that is not on this list, you must supply a **:hash-function** explicitly. **Note**: the **:test** and **:hash-function** interact closely, and must agree with each other.

:size An integer representing the initial size of the table. The table will be made large enough to hold this many entries without growing.

:area If **:area** is **nil** (the default), the ***default-cons-area*** is used. Otherwise, the number of the area that you wish to use. This keyword is a Symbolics extension to Common Lisp.

:hash-function Specifies a replacement hashing function. The default is based on the **:test** predicate. This keyword is a Symbolics extension to Common Lisp.

:rehash-before-cold Causes a rehash whenever the hashing algorithm has been invalidated, during a Save World operation. Thus every user of the saved world does not have to waste the overhead of rehashing the first time they use the table after cold booting.

For **eq** tables, hashing is invalidated whenever garbage collection or world compression occurs because the hash function is sensitive to addresses of objects, and those operations move objects to different addresses. For **equal** tables, the hash function is sensitive to addresses of some objects, but not to others. The table remembers whether it contains any such objects.

Normally a table is automatically rehashed "on demand" the first time it is used after hashing has become invalidated. This first **gethash** operation is therefore much slower than normal.

The **:rehash-before-cold** keyword should be used on tables that are a permanent part of your world, likely to be saved in a world saved by Save World, and to be touched by users of that world. This applies both to tables in Genera and to tables in user-written subsystems saved in a world.

This keyword is a Symbolics extension to Common Lisp.

:rehash-after-full-gc
Similar to **:rehash-before-cold**. Causes a rehash whenever the garbage collector performs a full gc. This keyword is a Symbolics extension to Common Lisp.

:entry-size This keyword is obsolete. **:entry-size** 2 is equivalent to **:number-of-values** 1. **:entry-size** 1 is equivalent to **:number-of-values** 0. This keyword is a Symbolics extension to Common Lisp.

:number-of-values Specifies the number of values associated with the key to be stored in the table. Currently, the only valid values are 0 and 1. If 0 is specified, the table functions return **t** for the value of the entry. This keyword is a Symbolics extension to Common Lisp.

:store-hash-code Specifies that the table system store the hash code for each key with the key. This keyword makes **make-hash-table** run

faster, since its use avoids the need to run a test function, unless the hash codes are the same. Use of this keyword increases the size of the table. Since **gethash** searches for keys equivalent to the supplied key under the supplied value of the **:test** argument, **:store-hash-code t** improves performance if the **:test** function pages or is slow. This keyword is a Symbolics extension to Common Lisp.

:mutating Turns mutation on and off. The overhead involved with specifying this keyword is relatively higher for small tables than for large ones. The default value is **t**. This keyword is a Symbolics extension to Common Lisp.

:initial-contents Set the initial contents for the new table. It can be either a table object to be copied, or a sequence of keys and values, for example:

```
'(KEY1 VALUE1 ... KEYn VALUEn)
```

This keyword is a Symbolics extension to Common Lisp.

:locking One of the following locking strategies: **:process**, **:without-interrupts**, **nil**, or a cons consisting of a lock and an unlock function. The default is to lock against other processes. This keyword is a Symbolics extension to Common Lisp.

:ignore-gc By default, if the hash function is sensitive to the garbage collector, the table is protected against GC flip. If you supply this keyword, the table is not protected.

If the hash function utilizes the address of a Lisp object that might be changed by the GC, the hash function must recompute the hash code if that address is changed. **:ignore-gc** asserts that the hash function never uses such addresses, so that it need not recompute the codes. The default depends on the hash function: if it's one of a small set of functions that Lisp knows do not depend on addresses, this defaults to **t** (meaning yes, it can ignore the GC). Otherwise, it chooses **nil**, which is always safe. **t** might make your program run faster (avoiding rehashes at GC time) but might also break your program (if the hash function depends on address values). This keyword is a Symbolics extension to Common Lisp.

:gc-protect-values The default is **t**. If **nil**, table entries are automatically deleted if a value becomes unreachable other than through the table. This keyword is a Symbolics extension to Common Lisp.

:growth-factor A synonym for **:rehash-size**. If the keyword is an integer, it is the number of entries to add, and if it is a floating-point number, it is the ratio of the new size to the old size. If the value is neither an integer or a floating-point number, an error is signalled. This keyword is a Symbolics extension to Common Lisp.

:growth-threshold A synonym for **:rehash-threshold**. If it is an integer greater than zero and less than the **:size**, it is related to the number of entries at which growth should occur. The threshold is the current size minus the **:growth-threshold**. If it is a floating-point number between zero and one, it is the percentage of entries that can be filled before growth will occur. If the value is neither an integer or a floating-point number, an error is signalled. This keyword is a Symbolics extension to Common Lisp.

:rehash-size The growth factor of the table when it becomes full. If the value of the keyword is an integer, it is the number of entries to add, and if it is a floating-point number, it is the ratio of the new size to the old size. If the value is neither an integer or a floating-point number, an error is signalled.

:rehash-threshold How full the table can become before it must grow. If it is an integer greater than zero and less than the value of **:size**, it is related to the number of entries at which growth should occur. The threshold is the current size minus the **:growth-threshold**. If it is a floating-point number between zero and one, it is the percentage of entries that can be filled before growth will occur. If the value is neither an integer nor a floating-point number, an error is signalled.

If you are using CLOE, **zl:make-hash-table** returns a newly created hash table with *size* entries. Argument *test* must be *eq*, *eql* or *equal* expressed as either symbols or as the function-quoted objects. Argument *rehash-size* can be an integer that provides the number of entries to add, or a floating point number that indicates the portion of the previous size to grow the hash table. Argument *rehash-threshold* also may be an integer or floating point number, and indicates the maximum capacity of the hash table before it should grow.

```
(setq hash-table-1 (make-hash-table))

(setq hash-table-2
      (make-hash-table :size (* number-of-my-symbols 100)
                       :rehash-size 2.0
                       :rehash-threshold 0.8
                       :test 'eq))
```

Compatibility Note: The following keywords are Symbolics extensions to Common Lisp: **:area**, **:hash-function**, **:rehash-before-cold**, **:rehash-after-full-gc**, **:entry-size**, **:number-of-values: :store-hash-code:**, **:mutating**, **:initial-contents**, **:optimizations**, **:locking**, **:ignore-gc**, **:gc-protect-values**, **:growth-factor**, and **:growth-threshold**.

For a table of related items: See the section "Table Functions".

See the section "Hash Table Interface to the Table Management Facility".

Table Internals

More About Tables

Many types of table objects are available, but only four basic differences exist among them:

Mutability — You can make a table that changes internal representation at run time, with the **:mutating** keyword argument to **make-hash-table**. **:mutating t** is the default. 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 you have a representation that is efficient, 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, optimizations are 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 property list internal representations are optimized for **eq**, and tables with array internal representations have a **char-equal** optimization.

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, association list and set.

Locking — When the table is read or updated, 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 just 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 automatically locked against garbage collection by the table facility.

The facility can be changed and customized beyond the scope of the differences discussed here.

Improving the Performance of Table Flavors

Here is a list of ways to improve the performance of flavor tables:

- Set **:locking** to **nil** to improve table performance, if the table is only accessed on one process.

- Set **:number-of-values** to 0 to improve table paging performance if only the key is needed.

- Use the function **sys:page-in-table** to bring back into main memory any swapped pages in a flavor table that have been swapped out to disk. See the function **sys:page-in-table**.

- Use the function **sys:page-out-table** to take all of the swapped pages in a flavor table out to main memory. See the function **sys:page-out-table**.

- Use the macro **sys:with-table-locked** to lock a table around some specified code. See the function **sys:with-table-locked**.

Hash Functions

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.

sxhash provides what is called "hashing on **zl:equal**"; that is, two objects that are **zl:equal** are considered to be "the same" by **sxhash**. In particular, if two strings differ only in alphabetic case, **sxhash** returns the same object for both of them because they are **zl:equal**. The value returned by **sxhash** does not depend on the case of any strings. Therefore, **sxhash** is useful for retrieving data when two keys that are not the same object, but are **zl: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 objects, it lets the hash facility know, so that **gethash** rehashes the table based on the new object addresses.

For related information, see the section "Defining Hash Functions and Hash-Test Functions".

Defining Hash Functions and Hash-Test Functions

A hash-test function determines how to compare hash table keys. Common Lisp supports these three: **eq**, **eql**, and **equal**. Symbolics Common Lisp also supports the

following: **char-equal**, **char=**, **string-equal**, **string=**, **zl:equal**, **zl:string-equal**, **zl:string=**.

If you use any of these hash-test functions, then you don't need to write your own hash function. However, if you use a test function that the system doesn't know about, you need to write your own hash function.

A hash function takes one argument and returns an integer. If the returned value might change when the garbage collector runs, then the hash function must return a second value telling when to rehash the function. A hash function must return the same integer for equivalent keys, and it should try to return different integers for non-equivalent keys.

A hash-test function takes two arguments. It compares them for some definition of equivalency. It returns **nil** if they are not equivalent, and it returns a non-**nil** value if they are equivalent.

For example, assume you are building a knowledge-base system in which facts consisted solely of (*attribute object value*) triples. You might want to be able to hash on the attribute and the object to give yourself a pointer to the currently stored value. You wouldn't want to hash on the entire list because if you were asserting a new value, hashing on the new list wouldn't find the old value. You wouldn't want to hash on just the object, or just the attribute, because these buckets would get very large.

The following code supports the example described above:

```
(defvar triple-hash-table nil)

(defun triple-hash-function (triple)
  (multiple-value-bind (hash-first gc-first)
      (cli::xeqhash (first triple))
    (multiple-value-bind (hash-second gc-second)
        (cli::xeqhash (second triple))
      (values (logxor hash-first hash-second)
              (max gc-first gc-second)))))

(defun triple-equal-test (first-triple second-triple)
  (and
    (eq (first first-triple) (first second-triple))
    (eq (second first-triple) (second second-triple))))

(setq triple-hash-table
      (make-hash-table :hash-function 'triple-hash-function
                       :test 'triple-equal-test))
```

The following definition is taken from the source code:

```
(defsubst xeqhash (x)
  ;; Must compute gc-dependence before calling %pointer
  (let ((gc-dependence (gc-dependence x)))
    (values (sys:%pointer x) gc-dependence)))
```

Note that **sys:%pointer** takes an object and returns its address as a fixnum. You must consider the garbage-collection dependence only if your hash function uses **sys:%pointer**. The garbage-collection dependence must be calculated before the call to **sys:%pointer**.

The form (**sys:gc-dependence** *object*) returns a value corresponding to what level of garbage collection you need to worry about for the *object*. Returning this as the second value of the hash function indicates that the hash table must be rehashed if this level of garbage collection runs.

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 that requires a very small table of relatively static size, or a lookup table that 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, 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. Though these data types have different (incompatible) interfaces, they all store data in a tabular fashion.

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 (**adjoin union**), 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)
```

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.

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 principles remain the same.

The two major differences between vectors and sets are:

- The time it takes to access an element of a one-dimensional array is constant, whereas the time it takes to access an element of a set depends on the length of the set and where that element resides in the set.

- The time it takes to add new element to the front of a set is constant, whereas the time it takes to add a new element to the front of an array is proportional to the length of the array.

For more information on sequences and sequence functions, see the section "Sequences".

Lists as Tables

You can build more complex tables out of sets and vectors. Tables made from general lists, association lists, property lists, and arrays are examples of this.

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** 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 association list representation would be:

```
((heron . wader) (loon . diver) (eagle . raptor))
```

Given this association list, you could retrieve the class of any bird in the list.

Another type of table with two elements is the *property list*. The main difference between an association list and a property list 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.

Each property in a property list 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 **getf** function.

For more information on lists, see the section "Lists".

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.

For more information on arrays, see the section "Arrays".

Zetalisp Hash Tables

Zetalisp hash tables are an older implementation of tables which is being phased out in favor of the table management facility. Keep in mind that although they are still part of Genera, they 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 **zl:equal**.

You can create a new hash table using the predicate **eq** for comparisons of the keys with the function **zl:make-hash-table**. You can create a new hash table using the predicate **zl:equal** for comparisons of the keys 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
```

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 for keeping ordered tables in general, and for maintaining priority queues, in particular.

Heap Functions and Methods

:clear Removes all 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 Tests whether the heap is empty, returning **t** if is, otherwise **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.

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, but is considered obsolete. Changing over to the new facility is a very straightforward process.

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 (zl:make-equal-hash-table :size 20))
Old syntax:     (setq table (make-hash-table :test #'zl:equal :size 20))
Old syntax:     (setq table
                  (make-instance 'si:eq-hash-table :size 20))

New syntax:     (setq table (make-hash-table :test #'eq :size 20))
```

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)
```

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**

New Table Facility Functions Versus Zetalisp Functions

Zetalisp functions should be changed over to their Symbolics Common Lisp 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**

Table Functions in the CL Package with SCL Extensions

Here is the table function that has Symbolics Common Lisp extensions:

Function	*Extension(s)*
make-hash-table	optional arguments *:area, :hash-function, :rehash-before-cold, :rehash-after-full-gc, :number-of-values, :store-hash-code, :mutating, :initial-contents, :locking, :ignore-gc, :growth-factor, :growth-threshold*

Functions and Dynamic Closures

Functions

What is a Function?

Functions are the basic building blocks of Lisp programs. There are many different kinds of functions in Symbolics Common 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**.

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?".) 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".

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". 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 **(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".

(: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".

(: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 **defmacro**) and **zl:defunp**, 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.

Simple Function Definitions

See the section "Function-Defining Special Forms". Information on defining functions, and other ways of doing so, are discussed in that section.

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.

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**.

Kinds of Functions

There are many kinds of functions in Symbolics Common 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 "What is a Macro?".

Conceptually, the evaluator knows specially about all special forms (hence their name). However, the Symbolics Common 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.

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 "Functions for Formatting Lisp Code". It can also be opened up and examined with the usual functions for list-structure manipulation.

There are two kinds of interpreted functions: **lambda**s and **si:digested-lambda**s. 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". 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.

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.

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".

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 function. 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".

An instance is a message-receiving object that has some state and a table of message-handling functions (called *methods*). See the section "Flavors".

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".

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".

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 **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 "Function-Defining Special Forms".

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".

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 an 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.

Lambda-List Keywords

This section lists all the keywords that can appear in the lambda-list (argument list) of a function, a macro, or a special form. Some of these keywords are allowed in the lambda-list of all three of these, while others are only allowed in one; those are so indicated. Some of these keywords are obsolete and should not be used in new code.

Keyword	*Use*	*Restrictions*
&optional	Introduces optional arguments	None
&rest	Introduces rest arguments All supplied arguments are stored in a list. Caution: In Genera this is not guaranteed to be a "real" list. Under CLOE it is a real list unless you request that it be stack consed.	Only one arg.
&key	Separates positional and rest parameters from keyword parameters.	Parameters must be pairs.
&allow-other-keys		**&key** must also be used.
&aux	Separates arguments from auxiliary variables.	None

The following keywords work for macros defined by **defmacro** or **macrolet** only:

&body
&whole
&environment

The following keywords are obsolete, and not available in CLOE:

zl:&special
zl:&local
zl:&eval
zl:"e
zl:&functional
zl:"e-dontcare
zl:&list-of

For more information on how lambda-list keywords are treated: See the section "Evaluating a Function Form".

Although symbols with names prefaced by an ampersand, lambda-list keywords are not elements of the the keyword package . Unlike symbols which name parameters, these keywords indicate how to interpret arguments in function or macro calls.

Keywords **&body**, **&environment**, and **&whole** can only be used with **defmacro**. The other keywords can be used with **defun**, as well as **defmacro**. A list of lamda-list keywords follows:

- **&whole** When present, this keyword must be first in the lambda-list or component lambda-list. In the following example, the variable is bound to the entire macro call form, or to the analogous form at the component level.

  ```
  (defmacro fred (&whole all x y)
    (list (if pred all) x y))
  ```

  The variable *all* is bound to the entire call

  ```
  (fred a b)
  ```

  but, if the macro definition and call are

  ```
  (defmacro fred (z (&whole all x y))
    (list (if pred all) x y z))

  (fred 6 (foo b))
  ```

  then variable *all* is bound to

  ```
  (foo b).
  ```

- **&environment** Used to explicitly pass an environment to explicit calls to macroexpand in a macro definition.

- **&optional** This keyword is followed by a list of variable names, or lists that include a variable name, an initialization form for the variable when an argument is not supplied, and an optional third element. The first element of the list or the variable name is an optional parameter bound to a supplied value in the argument list, or initialized to **nil** (or the second component of the list when no argument is supplied). The third element of the list is a parameter with a not nil initialized value (in the event an argument was already supplied as the first parameter name in the list).

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).

documentation *name* &optional *(type* **'defun***)*
Finds the documentation string of the symbol, *name*, which is stored in various different places depending on the symbol type.

debugging-info *function*
Returns the debugging info alist of *function*.

arglist *function* &optional *real-flag arglist-finder*
Returns a representation of the lambda-list of *function*.

args-info *fcn*
Returns an integer called the "numeric argument descriptor" of the *function*, which describes the way the function takes arguments.

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**, **defconstant**, **zl:defconst**
defflavor	flavor	**defflavor**
defstruct	structure	**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.

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 Show Expanded Lisp Code, **mexp** or the Zmacs command Macro Expand Expression (c-sh-M) to look at the macro expansion of:

```
(defstruct (foo :conc-name) one two)

:Show Expanded Lisp Code (defstruct (foo :conc-name) one two)

(eval-when (eval compile zl:load)
  (record-source-file-name 'foo 'zl:defstruct)
  (record-source-file-name 'foo 'si:deftype)
  (when (zl:get 'foo 'dw:presentation-type-descriptor)
    (dw:check-type-redefinition 'foo 'defstruct))
  (defprop foo-two (foo . two) si:defstruct-slot)
  (sys:defsubst-with-parent foo-two (foo zl:defstruct) (foo)
    (aref foo 2))
  (defprop foo-one (foo . one) si:defstruct-slot)
  (sys:defsubst-with-parent foo-one (foo zl:defstruct) (foo)
    (aref foo 1))
  (sys:defsubst-with-parent make-foo (foo zl:defstruct)
                            (&key one two)
    ((lambda (#:g3153) (zl:aset two #:g3153 2)
             (zl:aset one #:g3153 1) #:g3153)
     (zl:make-array 3 ':named-structure-symbol 'foo)))
  (defprop foo
           (si:one :named-array nil
            ((one 0 nil si:%%defstruct-empty%% t nil foo-one)
             (two 1 nil si:%%defstruct-empty%% t nil foo-two)) t
            ((make-foo)) nil nil 2 nil foo nil 0
            (eval compile zl:load)
            nil foo- t nil nil foo-p copy-foo nil nil)
           si:defstruct-description)
  (defun copy-foo (si:x)
    (declare (sys:function-parent foo zl:defstruct))
    ((lambda (#:g3152) (zl:aset (aref si:x 1) #:g3152 1)
             (zl:aset (aref si:x 2) #:g3152 2) #:g3152)
     (zl:make-array 3 ':named-structure-symbol 'foo)))
  (defsubst foo-p (si:x)
    (zl:typep si:x 'foo))
  'foo)
```

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.

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**.

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**.

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**.

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**, **progv**, **let**, or any of several other special forms), and then use that environment elsewhere, even after the contour has been exited.

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))
        (zl: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 () x)**, or to a compiled function, as would **(function (lambda () 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:

```
(zl: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". See the section "Funargs and Lexical Closure Allocation".

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 **symbol-value**, 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.

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 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**.

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*.

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.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, use the Common Lisp equivalents of these functions.

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-closure**.

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**.

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**, **member** and **special-form-p** return non-**nil** values when the condition is true, while predicates such as **numberp**, **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 **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. For a full description of individual predicates, see the document *Symbolics Common Lisp Dictionary*.

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".

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.

sys:fixnump *object* Tests for fixnum.

Note: The following Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:bigp *object* Tests for bignum.

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.

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

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.

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 its name rather than itself (such as #\Line).

graphic-char-p *char* — Tests if *char* is a printing character.

standard-char-p *char* — Tests if *char* is one of the Common Common Lisp standard characters.

mouse-char-p *char* — Tests if *char* is a mouse-character representing the clicking of a mouse button.

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.

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.

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".

simple-string-p *object* — Determines if *object* is a simple string array (one with no fill pointer and no displacement), returning **t** if it is, and **nil** otherwise. Accepts any object as an argument.

string-char-p *char* — Determines if *char* can be stored into a thin string (that is, if it is a standard character), returning **t** if it can, and **nil** otherwise. Accepts a character argument only.

string-fat-p *string* — Determines if *string* is an array of fat characters, returning **t** if it can, and **nil** otherwise. Accepts a string argument only.

stringp *object* — Determines if if *object* is either type of string, returning **t** if it is, and **nil** otherwise. Accepts any object as an argument.

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.

listp *object* — Tests if *object* is a cons or the empty list.

locativep *object* — Tests if *object* is a locative.

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.

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 *type1* is definitely a subtype of *type2* (for exact return values see the dictionary entry).
typep *object type*	Tests if *object* is of type *type*.

Note: The following Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

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).

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 number is zero.

Note: The following Zetalisp predicate is included to help you read old programs. In your new programs, if possible, use the Common Lisp equivalent of this predicate.

zl:signp *test number* — Tests if the sign of *number* matches *test*.

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

Case-Sensitive Character Comparison Predicates

user::char////////= *char* &rest *more-chars*	Not the same.
char≠ *char* &rest *more-chars*	Not the same (same as **user::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>=**).

Case-Insensitive Character Comparison Predicates

These predicates test characters using a different ordering scheme that accounts for differences in font information, but ignores differences in bits attributes and case.

char-equal *char* &rest *more-chars* Like **char=**.

char-not-equal *char* &rest *more-chars* Like **user::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>=**.

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".

The keywords *:start1* **0** and *:start2* **0** 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 **nil**. 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 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.

string= *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if two strings are are identical in all character fields, including modifier bits, character set, character style, and alphabetic case; it is false otherwise.

string≠ *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if the characters in the two strings are not identical (same as **user::string////=**).

user::string////= *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
A synonym of **string≠**.

string< *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
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* (*:start2* **0**) *:end2*
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* (*:start2* **0**) *:end2*
A synonym of **string≤**.

string> *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

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* (*:start2* **0**) *:end2*

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* (*:start2* **0**) *:end2*

A synonym of **string≥**.

string-exact-compare *string1 string2* &key (*:start1* **0**) (*:start2* **0**) *:end1 :end2*

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*.

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.

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**.

Note: These Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:string-exact-compare *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2*

Returns a positive number is *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. Use the Common Lisp function **string-exact-compare**.

zl:string= *string1 string2* &optional *idx1 idx2 lim1 lim2*

Like **string=**, but returns **t** or **nil**.

zl:string≠ *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *lim1 lim2*

Like **string≠**, but returns **t** or **nil**.

zl:string< *string1 string2* &optional (*idx1* **0**) (*idx2* **0**) *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**.

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".

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** (the beginning of the string); the end arguments default to **nil** (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 of the first character at which the strings fail to match; this index is equivalent to the length of the common portion of the strings.

These predicates ignore the character fields for character style and alphabetic case for the comparison.

string-equal *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if two strings are are identical in all character fields, including modifier bits, character set, and character style; otherwise it is false. Case-insensitive version of **string=**.

string-not-equal *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Test if *string1* is not equal to *string2*. If the condition is satisfied, **string-not-equal** returns the position within the strings of the first character at which the strings fail to match. Case-insensitive version of **user::string////=**.

string-lessp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if the first characters that differ between *string1* and *string2* are **char<**, or if *string1* is a proper substring of *string2*. Case-insensitive version of **string<**.

string-greaterp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if the first characters that differ between *string1* and *string2* are **char>**, or if *string2* is a proper substring of *string1*. Case-insensitive version of **string>**.

string-not-greaterp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*
Tests if *string1* is less than or equal to *string2*. If the condition is satisfied, **string-not-greaterp** returns the position within the strings of the first character at which the strings fail to match. Case-insensitive version of **string<=**.

string-not-lessp *string1 string2* &key (*:start1* **0**) *:end1* (*:start2* **0**) *:end2*

Tests if *string1* is greater than or equal to *string2*. If the condition is satisfied, **string-not-lessp** returns the position within the strings of the first character at which the strings fail to match. Case-insensitive version of **string>=**.

string-compare *string1 string2* &key (*:start1* **0**) (*:start2* **0**) *:end1 :end2*

Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. 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. Returns a positive number if *string1* > *string2*, zero if *string1* = *string2*, and a negative number if *string1* < *string2*. Case-insensitive version of **sys:%string-exact-compare**.

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**.

These predicates ignore the character fields for character style and alphabetic case for the comparison.

Note: These Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:string-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings, returning **t** if they are equal and **nil** if they are not. Case-insensitive version of **zl:string=**. Use the Common Lisp function **string-equal**.

zl:string-not-equal *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings or substrings of them. 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*

Compares two strings using alphabetical order. 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>**. This compares two strings or substrings of them. Like **string-greaterp** but returns **t** or **nil**.

zl:string-not-lessp *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings, or substrings of them. Like **string-not-lessp** but returns **t** or **nil**.

zl:string-not-greaterp *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares two strings or substrings of them. Like **string-not-greaterp** but returns **t** or **nil**.

zl:string-compare *string1 string2* &optional *idx1 idx2 lim1 lim2*

Compares the characters of *string1* starting at *idx1* and ending just below *lim1* with the characters of *string2* starting at *idx2* and ending just below *lim2*. Case-insensitive version of **zl:string-exact-compare**. Use the Comon Lisp function **string-compare**.

Comparison-performing Predicates

eq *x y*	Tests for identical object (implementationally).
eql *x y*	Tests for identical object (conceptually).
equalp *x y*	Tests for similar objects.
neq *x y*	Not **eq**.
not *x*	Tests for the symbol **nil**.
null *object*	Tests for the empty list **nil** ().
alphalessp *string1 string2*	Like **string-lessp**, but also works on numbers, lists and other objects.

Note: The following Zetalisp predicates are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these predicates.

zl:equal *x y*	Tests for similar objects (differs from **equal** on arrays and characters)
zl:samepnamep *sym1 sym2*	Like **string=**; tests if the two symbols have **string=** printed representations

Predicates for Testing Bits in Integers

logbitp *index integer*

Returns t if *index* bit in *integer* (the bit whose weight is 2^{index}) is a one-bit.

logtest *integer1 integer2*

Returns **t** if any 1-bits in *integer1* are 1-bits in *integer2*.

Note: The following Zetalisp predicate is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this predicate.

zl:bit-test *x y* — Returns **t** if any 1 bits in *x* are 1 bits in *y*. Use the Common Lisp function, **logtest**.

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*.

Note: The following Zetalisp predicate is included to help you read old programs. In your new programs, if possible, use the Common Lisp equivalent of this predicate.

zl:instance-variable-boundp *var* — Tests if *var* is a bound instance variable

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.

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.

Time Predicates

time-elapsed-p *increment initial-time* &optional *(final-time* **(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*

Inline Functions and Macros

Inline Functions

An inline function is a function that compiles as an *open subroutine*. Thus, an inline function, also known as a substitutable function, is *open-coded* by the compiler. This means that when a call to some function that calls an inline function is compiled, *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. This is in contrast to normal functions, which compile as *closed subroutines*. For these, the compiler generates code to compute the values of the arguments and then applies the function to those values.

Whether a function is inline or not makes no difference in interpreted code; the difference is effected only in compiled code.

An inline function can be applied just as any function can be applied. The result of applying the inline function is the same as the result of applying a normal function.

Here is an example to illustrate the difference between inline functions and ordinary functions. Suppose there is a function called **square-sum** that calls another function, **square**:

```
(defun square-sum (a b)(square (+ a b)))
(defun square (x) (* x x))
```

If **square** is an ordinary function, then compiling **square-sum** produces code similar to the following:

```
0  ENTRY: 2 REQUIRED, 0 OPTIONAL
1  PUSH-LOCAL FP|0             ;A
2  BUILTIN +-INTERNAL STACK FP|1       ;B
3  CALL-1-RETURN #'SQUARE
```

Note the function call to **square**.

If **square** is an inline function, however, then compiling **square-sum** produces something like:

```
0  ENTRY: 2 REQUIRED, 0 OPTIONAL
1  PUSH-LOCAL FP|0            ;A
2  BUILTIN +-INTERNAL STACK FP|1      ;B      creating X(FP|2)
3  PUSH-LOCAL FP|2            ;X
4  PUSH-LOCAL FP|2            ;X
5  BUILTIN *-INTERNAL STACK
6  RETURN-STACK
```

Note that there is no function call here: the compiled code for **square-sum** incorporates the code that accomplishes **square**. The call to **square-sum** has been open-coded by substituting the inline function's definition into the code being compiled. This substitution is referred to as *expansion* of the inline function. Inline functions and macros are similar in this respect: they both result in expansions.

When to Use Inline Functions

Inline functions are used to produce faster code. This is because they avoid the overhead of function calls. Their use can also help produce code that is clearer and more readable than it would be without them. Compare forms A and B below:

```
A.

(defun square-sum (a b)(square (+ a b)))
B.

(defun square-sum (a b)(* (+ a b)(+ a b)))
```

A is more readable than B, and will run faster.

The disadvantages of inline functions are that:

- They cannot be detected by **trace** or the stepper in compiled code.
- When redefining an inline function, you must recompile every function whose source contains a call to it in order for the redefinition to take effect.

In general, however, if something can be implemented either as an inline function or as a macro, it is better to make it an inline function. Inline functions have the following advantages over macros:

- Inline functions can be passed as functional arguments. (For example, you can pass them to **mapcar**.)
- Inline functions can be applied to arguments, while macros cannot.
- The compiler binds the argument variables to the argument values of an inline function with **let**, so they get evaluated only once and in the correct order. When possible, the compiler then optimizes out the variables. Macros do not provide the simultaneous guarantee of argument evaluation order and generation of code with no unnecessary temporary variables.

Writing Inline Functions

The easy way to define an inline function is to use the SCL special operator **defsubst**. It is used just like **defun** and does almost the same thing. For example, the inline version of **square** is defined by

```
(defsubst square (x) (* x x))
```

The general syntax for **defsubst** is

(defsubst *name lambda-list . body*)

The argument, *name*, can be any function spec, but the inline expansion will occur only when *name* is a symbol.

You can define an inline function without relying on Symbolics Common Lisp extensions by using the Common Lisp inline proclamation in conjunction with a normal defun:

```
(defun square (x) (* x x))
(proclaim '(inline square))
```

See the declaration **inline**.

What is a Macro?

A *macro* is a special operator. It takes a form, or forms, and translates those forms into a new form. It then evaluates this new form. Macros can be defined by users. Like all special operators, macros do *not* evaluate their arguments.

Macros resemble inline functions in that, in compiled code, there is a substitution or *expansion* of forms. Macros, however, are much more powerful than inline functions, since they take effect (albeit differently) in interpreted code as well as in compiled code and because the way they and their arguments are treated is much more complicated.

When to Use Macros

Like inline functions, macros can be used to produce faster code than normal functions because they avoid the overhead of function calls. The primary use of macros, though, is *to define new syntactic constructs*. The fact that the arguments of macros are not evaluated makes this possible. Here is a simple example to demonstrate this:

Suppose you want to define your own version of **unless**. You might write this function:

```
(defun new-unless (condition result)
       (cond ((not condition) result)))
```

This definition works for simple cases: if **condition** evaluates to **nil**, the value of **result** is returned; otherwise **nil** is returned. For example,

```
(new-unless t 2) => nil
(new-unless nil 2) => 2
```

A more complicated case reveals problems:

```
(new-unless t (print 2))
```

This form returns **nil**, but before that it goes ahead and prints 2. This is not the correct behavior; the second argument should be evaluated only on condition. Macros solve this kind of problem.

Using macros to create new syntactic constructs allows you to extend Lisp and to create your own embedded languages within Lisp. Using macros also has the advantage of increased speed and clarity of code.

The disadvantages to using macros are:

- Code using macros can be difficult to debug because macros in compiled code cannot be detected by **trace** or **zl:step**.
- Because macros are not functions, they can not be used with **zl:apply**, **funcall**, or mapping functions.
- In order for a change in a macro to take effect, any function that uses that macro must be recompiled after that change is made.
- Macros, because Lisp treats them in a relatively complex way, can be tricky to use. (The section "Avoiding Common Macro-Writing Pitfalls" treats this issue.)

Because of these drawbacks, the usual order of preference for how to write a procedure in Lisp is:

1. Use a simple function for ordinary purposes and ease in debugging.
2. Use an inline function for extra speed where possible. Note that a procedure can be an inline function only if it has the exact semantics of a function, rather than a special form.
3. Use a macro for extra speed where an inline function is not possible, or for creating an extension to the language.

Writing Simple Macros

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.

Introduction to defmacro

The easiest way to create a macro is to use the macro **defmacro**, which is similar in syntax to **defun**. The syntax of **defmacro** is:

```
(defmacro name pattern . body)
```

Here is an explanation of the arguments:

- The macro's *name* is the symbol whose macro definition is being created. It can be any symbol.
- The *pattern* constitutes the macro's lambda-list (that is, its argument list). It can be anything made up of symbols and conses; it need not be a list, the way a function's lambda-list must.
- The forms in *body* specify the code into which the macro call is to expand. They constitute the body of the macro's expander function.

Here is how to write a simple macro using **defmacro**:

1. Figure out what the expansion of the macro should be. For example, the expansion of the **new-unless** macro should look like:

   ```
   (cond ((not condition) result))
   ```

2. Devise a body form that, when evaluated, produces the expansion you want. For example:

   ```
   (list 'cond (list (list 'not condition) result)))
                => (cond ((not condition) result))
   ```

   Do not quote the arguments *condition* and *result*, as they are not evaluated in the macro expansion and do not need quotes.

3. Write the macro using that body form. Here is one way to write the **new-unless** macro:

   ```
   (defmacro new-unless (condition result)
               (list 'cond (list (list 'not condition) result)))
   ```

Here is a sample call to the macro **new-unless**:

```
(new-unless (symbolp 2)(print 10))
```

causes the following things to happen:

- *condition* gets bound to the first argument, which is the list (symbolp 2). *result* gets bound to the second argument, the list (print 10). Note that this is not the same kind of binding that would occur with a function call; the argument forms are not evaluated.
- The body of the macro:

  ```
  (list 'cond (list (list 'not condition) result))
  ```

  evaluates with the above bindings in place. The result of that evaluation is the result of the macro expansion:

  ```
  (cond ((not (symbolp 2))(print 10)))
  ```

- The bindings used for the evaluation of the body are undone; they are not used in the evaluation of the result of the macro expansion.

- The macro expansion is evaluated and the result returned. In this case, 10 is printed and 10 is returned.

Basically, **defmacro** matches the symbols in *pattern* against variables within *body*. Non-**nil** symbols in *pattern* are bound to the parts of *body* to which they correspond. This is called *destructuring*.

Destructuring provides you with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Here the macro **destruc-ex** is defined:

```
(defmacro destruc-ex ((horn light-r light-l)
                      ((wheel-1 size-1) (wheel-2 size-2))
                      hood-ornament)
 ...)
```

destruc-ex is called this way:

```
(destruc-ex (h (car lights) (cdr lights))
            ((w1 (measure-size w1)) (w2 (measure-size w2)))
            winged-victory)
```

The **destruct-ex** macro causes the expansion function to receive the following values for its parameters:

Parameter	*Value*
horn	h
light-r	(car lights)
light-l	(cdr lights)
wheel-1	w1
size-1	(measure-size w1) . . .

defmacro installs the expander function as the global macro definition of *name* and returns *name* as its (the **defmacro** form's) value. When a macro that has been created this way is called, the expander function replaces the macro call with new code specified by the body of the expander function.

In the definition of **new-unless** given earlier in this section, it would be helpful if the body of the macro looked more like its expansion. The backquote-comma syntax allows you to write a macro so that its body clearly reflects its own expansion. See the section "Backquote-Comma Syntax".

For more information on **defmacro**: See the section "Using Advanced Features of **defmacro**".

Backquote-Comma Syntax

The backquote-comma syntax lets you create a template for macro expansion. This template makes the form of the body of a macro clearly reflect its own expansion. For example, consider the following macro definition:

```
(defmacro new-unless (condition result)
          (list 'cond (list (list 'not condition) result)))
```

It becomes much easier to understand when rewritten with the backquote-comma syntax:

```
(defmacro new-unless (condition result)
       `(cond ((not ,condition) ,result)))
```

Backquote (‘), when used before a form, has an effect similar to single-quote (’), in that it prevents the form that follows it from being evaluated. For example:

```
'(a b c) => (a b c)
`(a b c) => (a b c)
```

Backquote is different from single quote in that it causes items within the form that are preceded by commas to be evaluated. For example:

```
(setq b 1) => 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. (*Hint*: A way to remember which character is the quote and which is the backquote is to remember that the quote character, which is used more frequently, is placed on a key that is more easily accessible than the back-quote key when in the normal touch-typing position.)

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 parts of the form generated by the macro are constant, that is, expansion generates 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, that is, often being the "arguments" of the macro. These latter parts are the ones you put the commas in front of.

As an aside, note that you can use the backquote-comma syntax in other situations besides writing macros. For example, it is often used to make item lists for menus, where an item list consists of several items that remain constant along with a few things that will change. In general, in many places where you might write something like A, below, you could instead write B:

```
A.

(list first-arg second-arg)
```

you could instead write

```
B.
```

```
`(,first-arg ,second-arg)
```

Here are more examples of simple macros. The symbol ==> means "results in something like the following macro expansion." (The actual macro expansions produced by Common Lisp are slightly more complicated than those we show here as examples.)

A macro that lets you call **new-first** in place of **car**:

```
(defmacro new-first (the-list)
          `(car ,the-list))

(new-first some-list) ==> (car some-list)
```

A macro that translates a form resembling (**addone** *x*) into (+ 1 *x*):

```
(defmacro addone (symbol)
          `(+ 1 ,symbol))
```

A macro that translates (**increment** *x*) into (**zl:setf** *x* (1+ *x*)):

```
(defmacro increment (symbol)
          `(setf ,symbol (1+ ,symbol)))

(increment a) ==> (setq a (1+ a))
```

(No, that's not a typo. **setq** is really what it expands into.)

Finally, here is a macro that creates an iteration construct that increments a variable by one until it exceeds a limit (like the FOR statement of the BASIC language). The syntax of this construct is

```
(for a 1 100 (print a) (print (* a a)))
```

We want it to expand into something like

```
(do a 1 (1+ a) (> a 100) (print a) (print (* a a)))
```

Here is the macro definition, using backquote-comma syntax:

```
(defmacro for (var lower upper &body body)
   `(do ,var ,lower (1+ ,var) (> ,var ,upper) ,@body))
```

As an exercise, you might try writing this same macro without the backquote-comma syntax. Your answer should look something like:

```
(defmacro for (var lower upper &body body)
   (cons 'do
      (cons var
         (cons lower
            (cons (list '1+ var)
               (cons (list '> var upper)
                  body))))))
```

You can see how much easier the backquote-comma syntax makes this.

Remember that the *pattern* argument need not be a list. In the above example, the pattern was a "dotted list", since the symbol **&body** was supposed to match the **cddddr** of the macro form. Suppose you wanted a different syntax, say:

```
(for a (1 100) (print a) (print (* a a)))
```

then you could accomplish the change in the macro simply by modifying the pattern of the **defmacro** above, as:

```
(defmacro for (var (lower upper) &body body)
          `(do ,var ,lower (1+ ,var) (> ,var ,upper) ,@body))
```

For more information: See the section "Extensions to the Backquote-Comma Syntax".

Extensions to the Backquote-Comma Syntax

When an at-sign, @, follows a comma in a backquoted list (**,@**) it means that the following item evaluates to a list that is to be spliced into the current list. For example, if **a** is bound to (**x y z**), then:

```
`(1 ,a 2) => (1 (x y z) 2)
```

But:

```
`(1 ,@a 2) => (1 x y z 2)
```

You can use this construct, for example, to rewrite a function that takes **&rest** arguments so that, instead of a list, it takes any number of arguments. That is, you can rewrite a function whose syntax is:

```
(function &rest argument-list)
```

as a macro:

```
(defmacro new-function (&rest argument-list)
          `(function ,@list-of-arguments))
```

which can be called with the syntax:

```
(new-function arg1 arg2 arg3 arg4 ...)
```

When a dot . follows a comma in a backquoted list (**,.**) it means the same thing as the @, except in this case the list to be inserted can be destructively modified. That is, if ,@ can be thought of as generating a call to the **append** function, then ,. can be thought of as generating a call to **nconc**. As with other types of destructive modification, use of the ,. syntax can produce more efficient code, but it is dangerous.

Writing More Complicated Macros: Common Techniques

Using Advanced Features of defmacro

defmacro can have one or more declarations between the *pattern* argument and the *body* argument. For more information, see the section "Declarations"

defmacro also takes an optional documentation string between *pattern* and *body*. This string is attached to the macro's *name*, and serves as the documentation string. You can see the documentation string of any macro (or symbol) by using the function **documentation**, which returns the documentation string stored with the macro. (See the function **documentation**.)

For example, suppose you define:

```
(defmacro for (var lower upper . body)
  "Iteration construct with syntax (for a 1 100 (do-body-things))"
  `(do ,var ,lower (1+ ,var) (> ,var ,upper) ,@body))
```

then when users do:

```
(documentation 'for 'function)
```

they see the documentation string:

```
Iteration construct with syntax (for a 1 100 (do-body-things))
```

The documentation string and any declarations present can be in any order, unless there are no body forms in your macro. In this case, the documentation string must be followed by at least one declaration.

defmacro destructures all levels of patterns in a consistent way. The inside patterns can also contain **&**-keywords. **defmacro** checks the lengths of the pattern and subform for matching. (See the special form **destructuring-bind**.)

This behavior exists for all of **defmacro**'s parameters, except **&environment**, **&whole**, and **&aux**.

A **defmacro** *pattern* argument can contain the lambda-list keywords **&optional**, **&rest**, **&key**, **&allow-other-keys**, **&aux**, **&body**, **&whole**, and **&environment**. For **&optional** and **&key** parameters, initialization forms and "**supplied-p**" parameters can be specified, just as for **defun**.

Here are descriptions of the keywords unique to macro definition:

&body

This is identical in function to **&rest**, but it tells the pretty printer to indent the remainder of the form as a body. You can use either **&rest** or **&body** in one sublist, but you cannot use both in one macro. This, for example, is permissible:

```
((x &rest y) &body z)
```

&whole

This is followed by a single variable, *var*, that is bound to the entire macro-call form. *var* is the value that the macro-expander function receives as its first argument. **&whole** is allowed only in the top-level pattern, not in inner patterns. **&whole** and its variable must appear first in *pattern*, before any other parameter or keyword.

&environment

This is followed by a single variable, *env*, 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. *env* should be used only with the functions **macroexpand** and **macroexpand-1** for the sake of any local macro definitions that the **macrolet** construct may have established within that lexical environment. See the section "Controlling Macro Expansion" and the special form **macrolet**. **&environment** is allowed only in the top-level pattern, not in inner patterns. It is useful primarily in cases where a macro definition must explicitly ex-

pand any macros in a subform of the macro call before computing its own expansion. Here is an example:

```
(defmacro cxx (x) `(car ,x))
  => cxx

(defmacro car-or-cdr (any-list &environment env)
  (macroexpand `(cxx ,any-list) env))
  => car-or-cdr
(defun Stooges ()
  (macrolet ((cxx (x) `(cdr ,x)))
    (car-or-cdr '(Larry Moe Curly))))
  => Stooges

(Stooges)
  => (Moe Curly)
```

When **Stooges** calls **car-or-cdr**, **car-or-cdr** uses the definition of **cxx** given in the **macrolet**, which is **car-or-cdr**'s lexical environment. If **&environment** and *env* are omitted, **car-or-cdr** uses the global definition of **cxx**:

```
(defmacro car-or-cdr (any-list)
  (macroexpand `(cxx ,any-list)))
  => car-or-cdr
(defun Stooges ()
  (macrolet ((cxx (x) `(cdr ,x)))
    (car-or-cdr '(Larry Moe Curly))))
  => Stooges

(Stooges)
  => Larry
```

Writing Macros That Expand into Multiple Forms

Ordinarily, a macro expands into only one form. This is inconvenient because often a macro is required to expand into several things, all of which should happen sequentially at run time. The **progn** special form takes care of this difficulty, since its use at the top level causes the compiler to consider all forms within the **progn** to be top-level forms.

To illustrate the use of **progn** in macros, here is an example. Suppose **defparameter** does not exist and you want to implement it as a macro. Your macro must do two things: declare the variable to be special and set it to its initial value. To keep the example simple, the new definition does only these two things and has no options. The call:

```
(defparameter a (+ 4 b))
```

should be equivalent to the two forms:

```
(proclaim '(special a))
(setq a (+ 4 b))
```

The macro definition

```
(new-defparameter (variable init-form)
   '(progn
      (proclaim `(special ,variable))
      (setq ,variable ,init-form)))
```

produces this same result whether interpreted or compiled; each subform is processed just as if it had appeared at the top level.

Here is another example that illustrates the use of **progn** and also illustrates how you can use Lisp to write a customized language for your own application. In this example, we create a macro that defines commands for a new embedded language. The macro lets you put documentation strings next to the code they document. This way, the code and documentation can be updated and maintained together in the same manner as Lisp **defun** and **defmacro**. The way 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.

The macro, called **define-my-command**, defines commands in an interactive user system. For each command, **define-my-command** provides a function that executes the command and a text string to be used as interactive online documentation. (This macro is a simplified version of a macro that is actually used in the Zwei editor.) In our system, 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 on a list. A typical call to **define-my-command** looks like this:

```
(define-my-command move-to-top
   "This command moves you to the top."
   (do ()
      ((at-the-top-p))
        (move-up-one)))
```

This expands 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))))
```

define-my-command expands into three forms: the first sets up the documentation string, the second puts the command name onto the list of all command names, and the third is the **defun** that actually defines the function. Note that **defprop**,

zl:push, and **defun** all happen when the file is loaded. (See the function **eval-when**.) The macro definition is:

```
(defmacro define-my-command (name doc-string definition)
  `(progn
     (defprop ,name ,doc-string help)
     (push ',name *command-name-list*)
     (defun ,name ()
            ,definition)))
```

Nesting Macros

We have seen how to write macros that define functions. Now consider writing macros that define other macros. To do this, we need to extend the backquote-comma syntax so that we can nest a backquote for a macro. This macro is defined inside a backquote used by the defining macro.

The following rules specify the evaluation of an item which is preceded by a comma, and is within nested backquotes.

1. A single comma always matches the innermost backquote. The form following the comma is evaluated once, when the form starting with the inner backquote is expanded.

2. A quote in front of a comma means to quote the result of the evaluation indicated by the comma.

3. Two commas matching two backquotes means evaluate once the form following the comma when the form starting with the outer backquote is evaluated. The result of this evaluation is then evaluated when the form starting with the inner backquote is expanded.

4. Comma-quote-comma preceding an item means evaluate the item once, when the form starting with the outer backquote is expanded.

5. In general, commas match backquotes such that the leftmost comma matches the innermost backquote, and so on.

6. Unlike the evaluation of Lisp forms, macro expansion works from *outer* to *inner* expressions.

Here are examples to illustrate these rules.

Nested Backquotes and Single Commas (Rule 1)

Use this to create a macro-definer in the simplest case: no arguments are passed from the defining call through to the defined macro.

The defining macro takes an argument, *name*, and creates a new macro with that name.

```
(defmacro simplest-defstruct (name)
      `(defmacro ,name (x) `(aref ,x 0)))
      ↑          ↑         ↑      ↑
    this   ↔   this      this ↔  this
  backquote    comma  backquote  comma
```

Expanding a form like:

(**simplest-defstruct test**)

results in a form like:

```
(defmacro test (x)
          `(aref ,x 0))
```

Quoting a Comma (Rule 2)

Use this to pass an argument through to the defined macro, while preventing its evaluation.

The defining macro, given argument *mac-name*, produces a macro named *mac-name* that puts its own name on a list.

```
(defmacro make-a-list (mac-name)
    `(defmacro ,mac-name () '(list ',mac-name)))
```

Expanding (**make-a-list groceries**) results in:

```
(defmacro groceries ()
       `(list 'groceries))
```

and (**groceries**) returns (**groceries**).

Nested Backquotes and Double Commas (Rule 3)

Use this to put a list structure template inside a **defmacro** form. For example:

```
(defmacro put-on-list (a-list key element)
    `(setf ,a-list (nconc  ,a-list `((,,key . ,,element)))))
```

((,,key . ,,element)) within the back-quoted **setf** form is equivalent to **(,(cons ,key . ,element))**.

Here is another, peculiarly contrived, example in which the defining macro takes an argument, *m-name*, and creates a new macro with that name. The new macro assigns to its argument the *value of the symbol* used as the name of the macro.

```
                             This backquote↔these commas.
(defmacro make-mac-2 (m-name)      ↓        ↓       ↓
     '(defmacro ,m-name (m-arg) '(setq ,m-arg ,,m-name)))
      ↑         ↑                              ↑
This backquote↔this comma and ...              this comma.
```

Expanding a form like (**make-mac-2 test**) in a form like:

```
(defmacro test (m-arg)
          `(setq ,m-arg ,test))

(setq test 4) => 4
(test a) => 4
a => 4
```

Comma-Quote-Comma (Rule 4)

Use this to pass an argument through from the defining macro to the defined macro. The argument is evaluated only once *when the form starting with the outer backquote is expanded*.

The new macro prints a list containing its own name and the argument with which it is called:

```
                           This backquote↔these commas.
(defmacro make-mac-1a (name)  ↓         ↓       ↓
      `(defmacro ,name (arg) `(print '(,',name ,arg ))))
      ↑          ↑                      ↑
   This backquote↔this comma and this comma.
```

Expanding a form like (**make-mac-1a test**) results in a form like:

```
(defmacro test (arg)
          `(print '(,'test ,arg)))
```

and the expansion of (**test a**) looks like:

```
(print '(test a))
```

and so forth.

Quote-Comma-Quote Syntax (Rules 5 and 6)

Use this to pass an argument through from the defining macro to the defined macro. The argument is *not* evaluated.

In the following example, a defining macro creates another macro whose operation is to create a list that includes its own name and its argument. This illustrates the rules 5 and 6:

```
                          This backquote↔these commas.
(defmacro defmac (name)       ↓       ↓        ↓
      `(defmacro ,name (arg) `(list ',',name ',arg )))
      ↑          ↑                    ↑
  This backquote↔this comma and this comma.
```

Expanding a form like `(defmac test)` results in a form like:

```
(defmacro test (arg)
          `(list 'test ',arg))
```

and the expansion of `(test a)` looks like:

```
(list 'test 'a)
```

Finally,

```
(test a) => (test a)
```

Writing Macros to Surround Code

When you want the evaluation of a piece of Lisp code to happen in a specified context, you can place a macro "around" that piece of code. This is called a *surrounding macro*. You write the macro so it specifies the context. The following is a simple example:

Suppose we want to evaluate a number of forms such that any output is written with a specified base. Here is a macro that executes the forms within its body in the desired context:

```
(defmacro with-output-in-base ((base-form) &body body)
    '(let ((base ,base-form))
        ,body))
```

A call such as:

```
(with-output-in-base (*default-base*)
            (print x)
            (print y))
```

Expands to:

```
(let ((base *default-base*))
          (print x)
          (print y))
```

The preceding example is too simple to be useful, but it does illustrate the style of surrounding macros.

Here are some style conventions for macros that surround code:

- Begin the macro with **with-**, in the style of such special forms as **with-open-file** and **zl:with-output-to-string**. Macros so named mean "do this *with* the following things true."

- Put the parameters for the macro in a list that is the first subform of the macro. The rest of the subforms in the macro make up a body of forms that are evaluated sequentially with the last one returned. In the example above, *base-form* is the sole parameter, and it appears as the single element of a list that is the first subform of the macro. 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 specify default parameters.

- Use the **&body** keyword in the **defmacro** form to tell the editor how to indent the macro. See the lambda list keyword **&body**.

You should write the macro in such a way that everything is cleaned up appropriately whether control leaves the macro by the last form's returning or by a nonlocal exit (that is, with something doing a **throw**). In the example above, there is no problem because nonlocal exits undo lambda bindings. More complicated cases require the use of an **unwind-protect** form. 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 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. Whenever control leaves **using-resource**, the allocated resource is deallocated.

Nesting Macros That Surround Code: macrolet and compiler-let

You can write a macro that is intended to be invoked *only within a specified environment*. The way you guarantee that this special environment will be in effect is by enclosing the macro inside a code-surrounding macro written especially for the purpose. Very often in this kind of structure you want the surrounded macro to be able to use variables associated with the surrounding macro. The special form **macrolet** makes the creation of this kind of structure easy, as the following simple example shows.

We are going to write two macros: the outer, surrounding one is called **with-collection** and the inner, included one is called **collect**. **with-collection** just has a body, whose forms it evaluates sequentially; **collect** takes one subform, which it evaluates. **with-collection** returns a list of all of the values that were given to **collect** during the evaluation of **with-collection**'s body. For example:

```
(with-collection
   (dotimes (i 5)
      (collect i)))

        => (0 1 2 3 4)
```

To take the first step in writing a macro, write down what the expansion should look like:

```
(let ((#:g0005 nil))
   (dotimes (1 5)
      (push i #:g0005))
   (nreverse #:0005))
```

From this expansion, we can see that **with-collection** should be defined by:

```
(defmacro with-collection (&body body)
   (let ((var (gensym)))
     `(let ((,var nil))
        ,@body
        (nreverse ,var))))
```

We might try to define **collect** by:

```
(defmacro collect (argument)
          `(push ,argument ,var))
```

But this will not work because this definition of **collect** is making use of **var** as a free variable, and **var** is unbound. The expander function of **with-collection** does bind **var** but then unbinds it when the expansion is finished before the **collect** form is expanded.

Since **collect** is only required to be defined within the body of **with-collection**, and it needs to use the variable **var**, we can use a **macrolet** form to define **collect**. **macrolet** is similar to **flet** but it defines local macros using the same format and keywords as **defmacro**; that is to say, wrapping a **macrolet** around some body forms within an enclosing definition is similar to using a **let** with the additional specification, "Evaluate these forms using the macro definition(s) listed within this **macrolet**."

The example becomes:

```
(defmacro with-collection (&body body)
   (let ((var (gensym)))
     `(let ((,var nil))
        (macrolet ((collect (argument)
           `(push ,argument ,',var)))
                ,@body)
        (nreverse ,var))))
```

Note that the *body* of **with-collection** is evaluated within the **macrolet**. Also, note that the backquote-comma syntax is used, since we are defining a macro within a macro.

A good practice to follow when writing local macros like **collect** is to define a global macro with the same name nearby in the source. Doing so makes your macro known to the editing tools and gives you a better error message if the macro is used in the wrong place. The global macro will of course be shadowed by the local macro when it is used correctly within its surrounding form. Here is a global macro to accompany the above example:

```
(defmacro collect (argument)
  (compiler:warn () "~S used outside of ~S"
                 'collect 'with-collection)
 `(ferror "~S used outside of ~S"
          `(collect ,,argument) 'with-collection))
```

The message for misuse of **collect** comes out both at compile time and at run time.

The macro-expander functions of **macrolet** are closed in the global environment; that is, no variable or function bindings are inherited from any environment. This 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 man-

agement must be utilized if this is done. See the section "Lexical Environment Objects and Arguments".

Here is another way to write **with-collection** and **collect**. This way is not as good as using **macrolet** because it introduces an unnecessary global variable and allows use of **collect** outside of **with-collection**, but it may prove instructive:

```
(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*))
```

The **compiler-let** form ensures that these macros work correctly when they are compiled. **compiler-let** tells the compiler to bind the variables it specifies — in this case ***collect-variable*** and the variable created by **zl:gensym** — and to compile its body with these bindings in effect. Thus the compiler works in this case just the same as the interpreter would.

Writing Macro-Expander Functions

A macro-expander function is a Lisp program like any other Lisp program, so it can benefit in all the usual ways by being broken down into a collection of functions that do various parts of its work. If you find yourself writing a five-page expander function, you should probably try to break your function down into modular parts. Several features of Symbolics Common Lisp, including flavors and the **loop** and **defstruct** macros, are implemented using very complex macros, which are broken down into modular functions. Studying the code of, for example, **defstruct** can suggest how to go about building a complex, modular macro.

A particular thing to note when writing macro-expander functions is that any functions that they use 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 Construction Tool helps keep these various files straight, compiling and loading things in the right order.

See the section "System Construction Tool".

Controlling Macro Expansion

The following functions and variable allow you to control expansion of macros. They are often useful for writing advanced macro systems or examining code that contains macros.

macroexpand-1 *macro-call* &optional *env dont-expand-special-forms* *Function*

If *macro-call* is a macro form, **macroexpand-1** expands it (once) and returns the expanded form and **t**. Otherwise, it returns *macro-call* and **nil**. The optional *env* environment parameter is conveys information about local macro definitions as defined via `macrolet`.

```
(defmacro nand (&rest args) `(not (and ,args)))

(macroexpand-1 '(nand foo (eq bar baz)(> foo bar)))

 ==> (not (and foo (eq bar baz)(> foo bar))) T

(defmacro and-op (op &rest args) `(,op ,args))

(macroexpand-1 '(and-op or (eq bar baz)(> foo bar)))

 ==> (or (eq bar baz) (> foo bar)) T
```

(See the section "Lexical Environment Objects and Arguments".)

Compatibility Note: The optional argument *dont-expand-special-forms*, is a Symbolics extension to Common Lisp, which prevents macro expansion of forms that are both special forms and macros. *dont-expand-special-forms* will not work in other implementations of Common Lisp including CLOE. See the variable ***macroexpand-hook***.

macroexpand *macro-call* &optional *env dont-expand-special-forms for-declares* *Function*

If *macro-call* is a macro form, **macroexpand** expands it repeatedly by making as many repeated calls to **macroexpand-1** as required until it is not a macro form, and returns two values: the final expansion and **t**. Otherwise, it returns *macro-call* and **nil**. The optional *env* environment parameter conveys information about local macro definitions that are defined via `macrolet`. (See the section "Lexical Environment Objects and Arguments".)

Compatibility Note: The optional argument *dont-expand-special-forms*, is a Symbolics extension to Common Lisp, which prevents macro expansion of forms that are both special forms and macros. *dont-expand-special-forms* will not work in other implementations of Common Lisp including CLOE.

```
(defmacro nand (&rest args) `(not (and ,args)))
```

```
(macroexpand '(nand foo (eq bar baz)(> foo bar)))

 ==> (not (and foo (eq bar baz)(> foo bar)))
```

The following example shows the probable results of three calls to **macroexpand-1** from within a call to `macroexpand`:

```
(defmacro and-op (op &rest args) `(,op ,args))

(macroexpand '(and-op or (eq bar baz)(> foo bar))) =

     (macroexpand-1 (and-op or (eq bar baz) (> foo bar)))
      ==> (or (eq bar baz) (> foo bar)) t

     (macroexpand-1 (or (eq bar baz) (> foo bar)))
      ==> (cond ((eq bar baz)) (t (> foo bar))) t

     (macroexpand-1 (cond ((eq bar baz)) (t (> foo bar))))
      ==> (if (eq bar baz) (eq bar baz) (> foo bar)) t

      ==> (if (eq bar baz) (eq bar baz) (> foo bar)) t
```

macroexpand-hook *Variable*

The value 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.

This special variable allows for more efficient interpretation of code, for example, by allowing caching of macro expansions. Such efficiency measures are unnecessary in compiled environments such as the CLOE runtime system.

Avoiding Common Macro-Writing Pitfalls

As powerful and convenient as macros are, writing them can be tricky. This section gives you some hints on how to avoid the most common problems.

Avoiding Problems with Backquote

Do not write programs that depend on the actual form resulting from evaluation of a backquoted form. Backquote makes no guarantees about how it does what it does. For example, when the reader sees `` `(a ,b c) `` it is actually generating a form such as `(list 'a b 'c)` The actual form generated might use **list**, **cons**, or

append, or — as is actually the case — it might use some special function so that **grindef** can figure out how to print out a backquoted form just the way it was typed in. More problematically, the reader might create constant forms that cause sharing of the list structure at run time, or 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)` that is, `'(r . nil)`, as opposed to just (**r**). Be careful that your program does not depend on a particular outcome.

Avoiding Name Conflicts

An accidental name conflict 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. The best way to avoid name conflicts is to use an uninterned symbol as the variable in the generated code. You can create an uninterned symbol either with **make-symbol** (the preferred way) or with **zl:gensym** (an older way). Here is an example of a typical name-conflict problem and its solution:

Suppose we want to write our own **zl:dolist** macro. Step one is to write down the expansion for a typical call, such as:

```
(new-dolist (element '(a b))
   (push element *big-list*)
   (foo-function element 3))
```

The expansion should look like:

```
(do ((operand '(a b) (cdr operand))
     (element))
    ((null operand))
  (setq element (car operand))
  (push element *big-list*)
  (foo-function element 3))
```

We write a macro that generates the above code in the obvious way, and this code works fine until a user happens to call our macro as follows:

```
(new-dolist (operand '(a b))
   (push element *big-list*)
   (foo-function operand 3))
```

Here the user has decided to name the looping variable **operand** rather than **element**. Expansion of this call results in:

```
(do ((operand '(a b) (cdr operand))
     (operand))
    ((null operand))
  (setq operand (car operand))
  (push operand *big-list*)
  (foo-function operand 3))
```

This does not work at all. It is not even a valid program because it uses the same variable in two different iteration clauses of **do**.

Or, suppose the user happens to use the macro this way:

```
(let ((operand nil))
   (dolist (element '(a b))
      (push element operand)
      (foo-function element 3)))
```

Then the expansion actually contains two variables named **operand**. The user means to refer to the outer one, but the generated code for the **zl:push** uses the inner one.

Here is the solution. Use the uninterned symbol **#:LIST**, generated by the **make-symbol** function, as the variable in the generated code. The expansion is now:

```
(do ((#:LIST '(a b) (cdr #:LIST))
     (element))
    ((null #:LIST))
  (setq element (car #:LIST))
  (push element *big-list*)
  (foo-function element 3))
```

Now we can write the macro:

```
(defmacro new-dolist ((var form) . body)
   (let ((dummy (make-symbol "LIST")))
      `(do ((,dummy ,form (cdr ,dummy))
             (,var))
            ((null ,dummy))
            ,@body)))
```

Be careful of the case implications involved in using the reader. For example, (**make-symbol "list"**) returns **#:|list|,** which might not be what you want.

Because many system macros use symbols whose print names begin and end with a dot for internal variables, you should not name your user variables using this convention. A name like **.object.** is meaningful for people reading generated code or looking at the state of computation in the debugger; this is why the system uses the convention. Before there was **make-symbol**, the alternative solution was to use **zl:gensym**, which returns a new, meaningless, internal symbol such as **#:g0005** every time it is invoked. Now, since you can give meaningful strings to **make-symbol** for your internal variable names, there is no need to resort to a **.xxx.**-type naming convention.

Avoiding prog-Context Conflicts

A problem can occur when you write a macro that expands into a **prog** (or a **do**, or something that expands into **prog** or **do**). If you use **prog** with **nil** (or use named **do**'s), **return** passes through the error and returns from the **prog** as it ought to. But, a way to avoid potential problems with **prog** and **do** is to rewrite your programs to use **block** instead. **block** evaluates each form in sequence and normally returns the (possibly multiple) values of the last form. See the special form **block**.

Avoiding 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** which 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 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. You should try to make special forms that are similar to function forms work that way too, for clarity and consistency.

The macro **once-only** makes it easier for you to follow the principle just explained. It is most easily explained by example. The way you write **test** using **once-only** is as follows:

```
(defmacro test (reference form &environment env)
  (once-only (form &environment env)
    `(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 describes what **once-only** does. Note, however, that you can easily use **once-only** simply by imitating the example above.

A **once-only** form looks like this:

```
(once-only (variable-name &environment environment)
  form1
  form2
  ...)
```

variable-name is a list of variables. **once-only** is usually used in macros where the variables are Lisp forms. **&environment** should be followed by a single variable that is bound to an environment representing the lexical environment in which the macro is to be interpreted. Typically this comes from the **&environment** parameter of a macro. The forms are a Lisp program that presumably uses the values of the variables to construct a new form to be the value of the macro. When a call to the macro that includes the **once-only** form is macroexpanded, the form produced by that expansion will be evaluated.

The macro that includes the **once-only** form will be macroexpanded. The form produced by that expansion is then evaluated. In the process, the values of each of the variables in *variable-name* are first inspected. These variables should be bound to subforms, that probably originated as arguments to the **defmacro** or similar form, and will be incorporated in the macro expansion, possibly in more than one place.

Each variable is then rebound 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 of the forms that are the values of variables in *variable-name* once, unless evaluation of the form has no side effects. At the same time, no unnecessary lambda-binding appears in the program. 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: **once-only** can be used only with an **&environment** keyword argument. If this argument is not present, a compiler warning will result.

For more information about using **once-only** with **&environment**: See the lambda list keyword **&environment**. Also, refer to the definitions of the macro defining forms: **defmacro**, **macrolet**, and **defmacro-in-flavor**.

```
(defmacro double (x &environment env)
  (once-only (x &environment env)
    `(+ ,x ,x)))
=> DOUBLE

(double 5)
==> (+ 5 5)

(double var)
==> (+ VAR VAR)

(double (compute-value var))
==> (LET ((#:ONCE-ONLY-X-3553 (COMPUTE-VALUE VAR)))
      (+ #:ONCE-ONLY-X-3553 #:ONCE-ONLY-X-3553))
```

Note that in the first three examples, when the argument is simple, it is duplicated. In the last example, when the argument is complicated and the duplication could cause a problem, it is not duplicated.

For information about avoiding problems with evaluation: See the section "Avoiding Multiple and Out-of-Order Evaluation".

once-only evaluates its subforms in the order they are presented. If it finds any form which is non-trivial, it rebinds the earlier variables to temporaries, and evaluates them first. In the following example, the order of evaluation is **x**, then **y**, even though the **y** appears before the **x** in the body of the **once-only**:

```
(defmacro my-progn (x y &environment env)
  (once-only (x y &environment env)
    ;; We willfully try to make it evaluate in the wrong order.
    `(progn ,y ,x))) => MY-PROGN

;;Macro expansion shows code that would be produced by the
;; once-only form in the macro.

(my-progn (print x) (setq x 'foo)) =>
(LET ((#:ONCE-ONLY-X-7614 (PRINT X)))
(PROGN (VALUES (SETQ X 'FOO)) #:ONCE-ONLY-X-7614))
```

In the next example, **once-only** evaluates **y**, then **x**, because **y** appears before **x** in **once-only**'s variable list. In actuality, this style is an example of poor programming practice as it is confusing. Always list variables in the order in which the forms they are bound to appear in the source that produced them. In a macro, this is normally the order they appear in the macro's argument list.

```
(defmacro backward-progn (x y &environment env)
  (once-only (y x &environment env)
    ;; We willfully try to make it evaluate in the wrong order.
    ;; But this time we tell once-only to evaluate y before x.
    `(progn ,y ,x))) => BACKWARD-PROGN

    (backward-progn (print x) (setq x 'foo)) => FOO
                                                FOO

    (PROGN (VALUES (SETQ X 'FOO)) (VALUES (PRINT X))) => FOO
                                                         FOO
```

Caution: A number of system macros, **zl:setf** for example, fail to follow this convention. Occurrences of unexpected multiple evaluation and out-of-order evaluation are possible. This implementation 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.)

Special Kinds of Macros

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* *Special 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. Then, 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 (gethash self x-hash-table)
```

To simulate setting a new value for **x**, you could use (**setf** x *value*), which would expand into (**setf** (**gethash self x-hash-table**) *value*).

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.

Use the special form **deflambda-macro** to deal with lambda macros. **deflambda-macro** works like **defmacro** to provide easy parsing of the function into its component parts. It defines a lambda macro instead of a normal macro.

deflambda-macro *name pattern* &body *body* *Function*

Like **defmacro**, but defines a lambda macro instead of a normal macro.

name is the name of the lambda macro to be defined; it can be any function spec. See the section "Function Specs". The *pattern* can be anything made up out of sym-

bols and conses. It is matched against the body of the lambda 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 b*ody*.

body is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

Here is an example of **deflambda-macro** used to define a lambda macro:

```
(deflambda-macro ilisp (arglist &rest body)
   `(lambda (&optional ,@arglist) ,@body))
```

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))
```

deffunction *fspec lambda-type lambda-list* &body *rest* *Special Form*

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.

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 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.

zl:displace *form expansion* *Function*

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") 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 *name pattern* &body *body* *Macro*

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.

Finding Out About and Debugging Macros

Expanding Macros

Sometimes a program bug appears to stem from unexpected behavior by a macro. Seeing how a macro form expands can help find the bug. To be sure that a macro does what you want it to, you might also want to create and expand a macro form soon after defining the macro and compiling the definition.

In Zmacs, you can use the following commands:

Macro Expand Expression (c-sh-M)
: Expands the macro form following point. Does not expand subforms within the form.

Macro Expand Expression All (m-sh-M)
: Expands the macro form following point and all subforms within the form.

Without a numeric argument, these commands type their results in the typeout window; with a numeric argument, the commands pretty-print their results in the buffer immediately after the expression.

You can also expand macros with the following special form:

(mexp) Enters a loop: prompts for a macro form to expand, expands it, and prompts for another macro form. Exits from the loop on END.

Macro expansion is particularly useful when trying to understand the workings of a macro that someone else wrote, such as a system macro. For example:

```
(mexp) =>
Type End to stop expanding forms
```

```
Macro form → (define-presentation-type Pinocchio ()
                        :abbreviation-for string) →
(ZL:LOCAL-DECLARE ((SYS:FUNCTION-PARENT PINOCCHIO
                                 DEFINE-PRESENTATION-TYPE)
                   (DW::PRESENTATION-TYPE-ARGLIST PINOCCHIO NIL))
  (DW::INITIALIZE-PRESENTATION-TYPE 'PINOCCHIO :ARGLIST 'NIL)
  (DEFTYPE PINOCCHIO NIL
    '(DW::PRESENTATION-ONLY-TYPE PINOCCHIO))
  (DEFUN (DW::PRESENTATION-FUNCTION PINOCCHIO
             DW::DATA-TYPE-EQUIVALENT) (#:TYPE)
    (DECLARE (COMPILER:DO-NOT-RECORD-THESE-MACROS
                 DW:WITH-PRESENTATION-TYPE-ARGUMENTS))
    (DW:WITH-PRESENTATION-TYPE-ARGUMENTS (PINOCCHIO #:TYPE)
      STRING))) →
(COMPILER-LET ((SYS:LOCAL-DECLARATIONS
                (APPEND '((SYS:FUNCTION-PARENT PINOCCHIO
                              DEFINE-PRESENTATION-TYPE)
                          (DW::PRESENTATION-TYPE-ARGLIST PINOCCHIO
                              NIL))
                        SYS:LOCAL-DECLARATIONS)))
  (DW::INITIALIZE-PRESENTATION-TYPE 'PINOCCHIO :ARGLIST 'NIL)
  (DEFTYPE PINOCCHIO NIL
    '(DW::PRESENTATION-ONLY-TYPE PINOCCHIO))
  (DEFUN (DW::PRESENTATION-FUNCTION PINOCCHIO
             DW::DATA-TYPE-EQUIVALENT) (#:TYPE)
    (DECLARE (COMPILER:DO-NOT-RECORD-THESE-MACROS
                 DW:WITH-PRESENTATION-TYPE-ARGUMENTS))
    (DW:WITH-PRESENTATION-TYPE-ARGUMENTS (PINOCCHIO #:TYPE)
      STRING)))
```

Hints for Debugging Macros

Here are a couple of hints to make debugging easier:

- When writing long macros, you might find it helpful to capitalize the Lisp tokens to be returned and keep the active code of the macro in lowercase. For example:

  ```
  (defmacro round-sqrt (foo)
          `(ROUND (SQRT ,foo)))
  ```

  This practice helps you to distinguish between the two levels of code evaluation.

- With very complicated macros, you can simplify the debugging process by defining a helper function for the body of the macro. The macro simply calls the helper function, which can be debugged in the same manner as any other function. With this method, the above example becomes:

```
(defun helper (foo)
       `(ROUND (SQRT ,foo)))

(defmacro round-sqrt (foo)
       (helper foo))
```

Declarations

Declarations are optional Lisp expressions that provide the Lisp system, typically the interpreter and the compiler, with information about your program. With the exception of the **special** declaration, all declarations are only advisory. They do not affect the meaning of an otherwise correct program. The compiler uses them to provide error checking or to produce more efficient code. Declarations can also provide documentation in code.

The **special** declaration affects the meaning of a program by affecting the interpretation of variable bindings and references. All special (global) variables must be declared **special**.

The special operator **declare** is the most common mechanism for making declarations. Global declarations and declarations that a program computes are made with the function **proclaim**. The special operator, **zl:local-declare** should *not* be used for new code.

Operators for Making Declarations

declare &rest *ignore*
: Provides additional information (*declarations*) to the Lisp system (interpreter and compiler).

proclaim *declaration*
: Puts the declaration specifier *declaration* into effect globally.

locally &body *body*
: Makes local pervasive declarations.

the *type form*
: Declares that the value of *form* is of type *type*.

zl:local-declare *declarations* &body *body*
: Obsolete. Use **locally** instead.

Symbolics Common Lisp provides a form for removing declarations made with **proclaim**: **remove-proclaims** *fspec*.

Declaration Specifiers

declaration *name1 name2 ...*
: Proclaims *names* to be valid but non-standard declarations.

ftype *type function-name-1 function-name-2 ...*
Specifies that the functions *function-names* are of type *type*.

function *name arglist result-type1 result-type2 ...*
Equivalent to **ftype** *type function-name-1 function-name-2* but might be more convenient.

ignore *var1 var2 ...*
Specifies that bindings of the *vars* are never used.

inline *function1 function2 ...*
Specifies that calls to *functions* should be open-coded.

notinline *function1 function2 ...*
Specifies that the *functions* should not be open-coded.

optimize (*option1 value1*) (*option2 value2*) ...
Specifies that the *options* (**compilation-speed**, **safety space**, and **speed**) should be optimized according to *values*.

special *var1 var2 ...*
Specifies that *vars* are to be considered special.

type *type var1 var2 ...*
Specifies that the variables *vars* only take on values of type *type*.

Many forms, such as **defun**, **defvar**, and **defconstant**, 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 **defconstant** (and **zl:defconst**) tell the system that certain symbols are special.

Function-body Declarations

Function-body **declare** forms understand the following declarations. The first group of declarations can be used only at the beginning of a function body, for example, **defun**, **defmacro**, **defmethod**, **lambda**, or **flet**.

(arglist . *arglist*)
This declaration saves *arglist* as the argument list of the function, to be used instead of its lambda-list if c-sh-A or the **arglist** function need to determine the function's arguments. The **arglist** declaration is used both for documentation purposes and as information for the compiler.

Example:

```
(defun example (&rest options)
  (declare (arglist &key x y z))
  (lexpr-funcall #'example-2 "Print" options))
```

The compiler checks keyword arguments supplied in a function call against the keyword arguments accepted by the called function. As with checking the number of arguments in a function call, this checking does not work if the function call is earlier in the file or group of files than the definition of

the called function. If there is an **arglist** declaration, it is used in place of the actual lambda-list to determine what keywords are accepted, since often the declared lambda-list contains **&key** but the actual lambda-list contains just **&rest**. The variable **compiler:*inhibit-keyword-argument-warnings*** can be set to **t** to disable this checking, for example if you have a lot of declared arglists that are malformed.

(values . *values***)**

This declaration saves *values* as the return values list of the function, to be used if `c-sh-A` or the **arglist** function asks what values it returns. The **values** declaration is used purely for documentation purposes.

(sys:function-parent *name type***)**

Helps the editor and source-finding tools (like `m-.`) locate symbol definitions produced as a result of macro expansion. (The accessor, constructor, and alterant macros produced by a **defstruct** are an example.)

The **sys:function-parent** declaration should be inserted in the source definition to record the name of the outer definition of which it is a part. *name* is the name of the outer definition. *type* is its type, which defaults to **defun**. See the section "How Programs Manipulate Definitions".

(sys:downward-function)

This declaration, 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
    (maphash
      #'(lambda (key object)
          (declare (sys:downward-function))
          (when (magic-function key object item)
            (return-from search object)))
      *hash-table*)))
```

Here **maphash** 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".

(sys:downward-funarg *var1 var2 ...***)** or **(sys:downward-funarg *)**

This 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, **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.

The following group of declarations can be used at the beginning of any body, for example, a **let** body.

(special *sym1 sym2 ...***)**

The symbols *sym1*, *sym2*, and so on, are treated as special variables within the form containing the **declare**; the Lisp system (both the compiler and the interpreter) implements the variables using the value cells of the symbols.

(zl:unspecial *sym1 sym2 ...***)**

The symbols *sym1*, *sym2*, and so on, are treated as local variables within the form containing the **declare**.

Example:

```
(defun zl:print-integer (number zl:base)
  (declare (zl:unspecial zl:base))
  (when (≥ number zl:base)
    (zl:print-integer (floor number zl:base) zl:base))
  (tyo (digit-char (mod number zl:base) zl:base)))
```

(sys:array-register *variable1 variable2 ...***)**

Indicates to the compiler that *variable1*, *variable2*, and so on, are holding single-dimensional arrays as their values. Henceforth, each of these variables must *always* hold a single-dimensional array. The compiler can then use special faster array element referencing and setting instructions for the **aref** and **user::aset** functions. Whether or not this declaration is worthwhile depends on the type of array and the number of times that referencing and setting instructions are executed. For example, if the number of referencing instructions is more than ten, this declaration makes your program run faster; for one or two references, it actually slows execution.

(sys:array-register-1d *variable1 variable2 ...***)**

Indicates to the compiler that *variable1*, *variable2*, and so on, are holding single- or multidimensional arrays as their values, and that the array is going to be referenced as a one-dimensional array. Henceforth, each of these variables must *always* hold an array. The compiler can then use special faster array element referencing and setting instructions for the **sys:%1d-aref** and **sys:%1d-aset** functions. Whether or not this declaration is worthwhile depends on the type of array and the number of times that referencing and setting instructions are executed. For example, if the number of referencing instructions is more than ten, this declaration makes your program run faster; for one or two references, it actually slows execution.

Evaluation

Introduction to Evaluation

The following is a complete description of the actions taken by the evaluator, given a *form* to evaluate.

form	*Result*
A number	*form*
A string	*form*
A symbol	The binding of *form*. If *form* is unbound, an error is signalled. See the section "Variables". Some symbols can also be constants, for example: **t**, **nil**, keywords, and objects created with **defconstant**.
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, Keywords, and Variables". 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". 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". 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 "What is a Macro?".
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". 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". That section contains a basic explanation of functions. See the section "Multiple Values". The way functions can return more than one value is explained there. See the section "Functions". 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 "A Hook Into the Evaluator". Special forms are described throughout the documentation set.

Evaluating a Symbol

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 that has no binding in the lexical environment, you get back the contents of the symbol's value cell.

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".

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.

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". 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.

rotatef Exchanges the value of one generalized variable with that of another.

Note: The following Zetalisp macro is included to help you read old programs. In your new programs, if possible, use the Common Lisp equivalent of this macro.

zl:swapf Exchanges the value of one generalized variable with that of another.

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 *lexpr*s, 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 constant **lambda-list-keywords**.

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.

Binding Parameters to Arguments

When **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. **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, **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.

5. 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.

6. 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. 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 sig-

nalled, 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**.

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**.

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 **(symbol-value 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.

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 ((:foo foo-value)))`

This allows a keyword that the caller knows under the name **:foo**, without making the parameter shadow the value of a variable **foo**.

`(&key ((:foo foo-value) 10 foo-supplied))`

When the **foo** keyword is supplied, the default value of 10 is ignored and **foo-supplied** is bound to **t**. If the keyword is not supplied, **foo-value** is bound to 10 and **foo-supplied** is bound to **nil**.

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.

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 **copy-list**.

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 **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**.

Some Functions and Special Forms

Functions for Function Invocation

apply *function argument* &rest *arguments*
: Applies the function *function* to *arguments*.

funcall *fn* &rest *args*
: **(funcall** *fn a1 a2 ... an***)** applies the function *fn* to the arguments *a1*, *a2*, ..., *an*.

send *object message-name* &rest *arguments*
: Sends a message to a flavor instance.

lexpr-send *object message argument* &rest *arguments*
: Like **send**, except that the last argument should be a list. All elements of that list are passed as arguments.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

zl:apply *fn args*
: Applies the function *fn* to the list of arguments *args*.

zl:lexpr-funcall *function argument* &rest *arguments*
: This is the Zetalisp equivalent of the Common Lisp **apply** function.

zl:call *fn* &rest *alternates*
: Offers a very general way of controlling what arguments you pass to a function.

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.

Note: The following Zetalisp special form is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this special form.

zl:comment *form* — Ignores its form and returns the symbol **zl:comment**.

Special Forms for Sequencing

progn &body *body* — The *body* forms are evaluated in order from left to right and the value of the last one is returned.

prog1 *value* &rest *ignore* — Similar to **progn**, but it returns *value* (its *first* form) rather than its last.

prog2 *ignore value* &rest *ignore* — Similar to **progn** and **prog1**, but returns its *second* form. It is included largely for compatibility with old programs.

progv *vars vals* &body *body* — First evaluates *vars* and *vals*, then binds each symbol to the corresponding value, and finally evaluates *body*. This provides the user with extra control over binding.

progw *vars-and-vals* &body *body* — Like **progv** except it evaluates and binds the *vars-and-vals* sequentially.

Function for Explicit Evaluation

eval *form* — Evaluates *form*, and returns the result. Mainly useful in Lisp systems, not application programs.

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.

Multiple Values

Symbolics Common Lisp includes a facility by which the evaluation of a form can produce more than one value. In most Lisp function calls, multiple values are not used. However, 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 using **setq** to assign special variables to the extra values.

A function must request multiple values. If the calling function does not request multiple values, and the called function returns multiple values, only the first value is given to the calling function. The extra values are discarded. Special syntax is required both to *produce* multiple values and to *receive* them.

Functions can return as many values as the value of **multiple-values-limit**. In Symbolics Common Lisp **multiple-values-limit** is 128.

Primitives 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*.

Special Forms for Receiving Multiple Values

The special forms for receiving multiple values are **multiple-value-setq**, **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.

multiple-value-setq *(variable...) form*

Calls a function that is expected to return more than one value.

multiple-value-bind *(variable...) form body...*

Similar to **multiple-value-setq**, 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.

Note: The following Zetalisp special form is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this special form.

zl:multiple-value *(variable...)*

The Zetalisp name for **multiple-value-setq**.

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 setting a variable to the result of a form using **setq**, then returning the value of that variable might be made to pass multiple values by an optimizing compiler that realized that the setting 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. See the section "Special Forms for Receiving Multiple Values".

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". 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**, **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. **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, **return**ing through the **zl:*catch**.

Scoping

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))
```

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.

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, **sort** and **<**.

```
(defun number-sort (numbers)
  (sort numbers #'<))
```

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. **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, **sort** is passed a lexical closure of the internal lambda. **sort** calls this closure many times to do comparisons. When **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 **magic-sort**.

In this example,

```
(defun make-adder (x)
  #'(lambda (y) (+ x y))) => MAKE-ADDER

(setq adder-4 (make-adder 4))
  => #<SYS:LEXICAL-CLOSURE (LAMBDA # #) 61115544>
(funcall adder-4 5) => 9
```

the closure of the internal lambda is returned from the activation of **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.

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.

sys:downward-function Declaration

This declaration, 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
    (maphash
      #'(lambda (key object)
          (declare (sys:downward-function))
          (when (magic-function key object item)
            (return-from search object)))
      *hash-table*)))
```

Here **maphash** 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".

sys:downward-funarg Declaration

This 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, **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.

flet, labels, and macrolet Special Forms

flet *functions* &body *body* *Special Form*

Defines named internal functions. **flet** (**f**unction **let**) defines a lexical scope, *body*, in which these names can be used to refer to these functions. *functions* 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**.

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 **squared** of **triangle-perimeter**, and then to define another subroutine, **distance**. Note that since **distance** is defined within the scope of the first **flet**, it can use **squared**. **distance** is then called three times in the body of the second **flet**. The names **squared** and **distance** are not meaningful as function names except within the bodies of these **flet**s.

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".

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 **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 **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 **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**.

In the following example, **eql** is redefined to a more liberal treatment for characters. Note that the global definition of **eql** is used in the local definition (this would not be possible with **labels**). Note also that **member** uses the global definition of **eql**.

```
(flet ((eql (x y)
         (if (characterp x)
           (equalp x y)
           (eql x y))))
  (if (member foo bar-list)              ;uses global eql
    (adjoin 'baz bar-list :test #'eql)   ;uses flet'd eql
    (eql foo (car bar-list))))
```

labels *functions* &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, 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".

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))))
```

In the following example, we use **labels** to locally define a function that calls itself. If we instead use **flet**, an error will result because the call to **my-adder** in the body would refer to an outer (presumably non-existent) **my-adder** instead of the local one.

```
(defun example-labels (operand-a operand-b)
  (labels ((my-adder (accumulator counter)
             (if (= counter 0)
                 accumulator
                 (my-adder (incf accumulator) (decf counter)))))
    (my-adder operand-a operand-b)))

(example-labels 6 4)  => 10
```

macrolet *macros* &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. *macros* 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".

The following example of **macrolet** is for demonstration only. If the macro **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


(defun foo (x)
  (macrolet ((do-it (var n)
               `(case ,var
                  ,(do ((i 0 (+ i 1))
                        (l '()))
                       ((= i n)(nreverse l))
                     (push (list i (format nil "~R" i))
                           l)))))
    (do-it x 100)))

(foo 12) => "twelve"
```

The following example implements a macro to establish a context where items can be added to the end of list. This is similar to the way **push** adds to the beginning of a list. We use **macrolet** to ensure that **push-onto-end** has access to the pointer until the last cons of the list.

```
(defmacro with-end-push2 (list &body body)
  (let ((lastptr (gensym)))
    `(let ((,lastptr (last ,list)))
       (macrolet ((push-onto-end (val)
                    `(rplacd ,',lastptr
                             (setq ,',lastptr (cons ,val nil)))))
         ,body))))

(defun example-3 ()
  (let ((mylist (list 1 2 3))
        (a-list (list 'a 'b 'c 'd)))
    (with-end-push2 mylist
      (dolist (l a-list mylist)
        (push-onto-end l)))))

(example-3)
```

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".

Flow of Control

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".) There is also a very general construct to allow the conventional "goto" control structure, called **prog**.

A *conditional* construct allows a program to decide to 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.

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.

Symbolics Common Lisp also provides a coroutine capability and a multiple-process facility. See the section "Scheduler Concepts". There is also a facility for generic function calling using message passing. See the section "Flavors".

Conditionals

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. For a description of **if**'s incompatibility with Common Lisp: See the special form **if**.

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 in *body*; 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 in *body* 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**, otherwise **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.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp equivalents of these functions.

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**.

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. For information about scoping: See the section "Scoping".

Blocks and Exits Functions and Variables

block *name* &body *body*
: Evaluates each form in *body* in sequence and normally returns the (possibly multiple) values of the last form in *body*.

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*
: Controls the display of compiler messages for invalid formats of **return** and **return-from**.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp versions of this function.

zl:return-list *form*
: Like **return** except that the block returns all of the elements of *list* as multiple values. Note that the Common Lisp form `(return(values-list` *list*`))` is preferred.

See the section "Nonlocal Exits".

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.

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.

Iteration

Two basic iteration functions in Symbolics Common Lisp are **do** and **prog**. Another iteration macro is documented elsewhere: See the macro **loop**.

Iteration Functions

do *vars endtest* &body *body*
: 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* *vars endtest* &body *body*
: 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 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 specified.

prog (*vars-and-vals*) **&body** *body*
: 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**.

Note: The following Zetalisp functions are included to help you read old programs. In your new programs, where possible, use the Common Lisp versions of these functions.

prog* (*vars-and-vals*) **&body** *body*

Just like **prog**, except that the binding of the temporary initialization variables is done sequentially.

zl:do-named *block-name* (*vars*) (*endtest*) **&body** *body*

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 *block-name* (*vars*) (*endtest*) **&body** *body*

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.

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 information on scoping, see the section "Scoping".

For example:

```
(catch 'done
  (ask-database <pattern>
                #'(lambda (x) (when (nice-p x)
                                (throw 'done x)))))
```

See the section "Blocks and Exits".

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 *protected-form* 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**, and **:always**.

sys:without-aborts ([*identifier*] *reason format-args* ...) *body* ...
: Encloses code that should not be aborted. A **sys:without-aborts** 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**.

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.

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. (The function **map** is a special case, discussed elsewhere. See the section "Sequences".)

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.

Sometimes a **do** or a straightforward recursion is preferable to a mapping function; 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.

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.

Note: The following Zetalisp function is included to help you read old programs. In your new programs, use the Common Lisp equivalent of this function.

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  |   successive  |
                    |   sublists   |    elements   |
     ---------------+--------------+---------------+
         its own    |              |               |
         second     |     map      |     mapc      |
        argument    |              |               |
     ---------------+--------------+---------------+
       list of the  |              |               |
Returns   function  |   maplist    |    mapcar     |
         results    |              |               |
     ---------------+--------------+---------------+
       nconc of the |              |               |
         function   |    mapcon    |    mapcan     |
         results    |              |               |
     ---------------+--------------+---------------+
```

There are also functions (**zl:mapatoms** and **zl:mapatoms-all**) for mapping over all symbols in certain packages. See the section "Package Iteration".

You can also do what the mapping functions do in a different way by using **loop**. See the section "The **loop** Iteration Macro".

The loop Iteration Macro

The **loop** macro provides a programmable iteration facility.

The basic structure of a **loop** is:

```
(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. **do** is the keyword that 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 **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 (*epi-*

logue) code. Variables can be declared local to the loop. The features you can incorporate using **loop** are: deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The **loop** macro works identically in both Symbolics Common Lisp and CLOE. The Symbolics implementation of **loop** is an extension of the Common Lisp specification for this macro, as specified in Guy L. Steele's *Common Lisp*: *the Language*. The **loop** version of this macro allows its body to be a sequence of lists, whereas Common Lisp's version of loop does not.

Note that **loop** forms are intended to look like stylized English rather than Lisp code. They contain 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** covers:

- **loop** Clauses
- **loop** Synonyms

The advanced discussion of **loop** deals with the following topics:

- Destructuring
- The Iteration Framework
- Iteration Paths

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**

Clause	*Keywords*
Iteration-driving	**repeat**, **for**, **as**
Initialization bindings	**with**, **nodeclare**
Entrance and Exit	**initially**, **finally**
Side Effects	**do[ing]**
Accumulating Return Values	**collect[ing], nconc[ing]** **append[ing], count[ing]** **sum[ming], maximize** **minimize**
End Tests	**until**, **while**, **loop-finish** **always**, **never**, **thereis**
Conditionalization	**when**, **if**, **unless**
Miscellaneous	**named**, **return**

The dictionary entry for each individual keyword covers it in detail.

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**).

The following kinds of iteration are possible:

- Iteration in series and in parallel
- Joining iteration clauses with **and**
- Iterating with **repeat**
- Iterating with **for** and **as**

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 grouped in series by default, 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
          ...)
```

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
          ...)
```

Here is now evaluation in iteration-driving clauses works:

- Those expressions that are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase.

- Those expressions that are evaluated each time around the loop are evaluated in order in the body.

Iterating with Repeat

One common and simple iteration-driving clause is **repeat**, which causes a specified number of iterations through the loop.

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
```

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".

Iteration Keywords and for Clauses

Here are the iteration keywords and all the varieties of **for** clauses. Optional parts are enclosed in braces. 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 the number of times specified by the value of *expression*. *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*}
Iterates *upward*. 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*}
Iterates *downward*. 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*} = *expr1* **then** *expr2*
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** *expr1* **then** *expr2*
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 for **loop**".

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
```

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.

Entrance and Exit in loop

To introduce initialization *(prologue)* code in **loop** use the keyword **initially**.

For example:

```
(loop for x from 1 to 4
      initially (princ "let's count... ")
      doing (princ x)) => let's 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 "let's count... ")
      doing (princ x)) => 1234let's count...
NIL
```

Entrance and Exit Keywords for loop

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.

Side Effects in loop

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.

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. (The optional argument, *data-type*, is currently ignored.) 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 **nreverse**s 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)))
```

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.

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.

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.

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 to something other than **nil** 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 to something other than **nil**.

thereis *expr* — If *expr* evaluates to something other than **nil**, 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.

loop Conditionalization

The keywords **when** 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**".

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.

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.

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))
```

Conditionalizing the return Clause

The **return** clause is useful with the conditionalization clauses. It causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is, the two clauses below are equivalent:

when *expr1* **return** *expr2*

when *expr1* **do (return** *expr2***)**

Conditionalizing an Aggregated Boolean Value Clause

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause iteration to terminate early from being 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))
```

In this case, the **extra-special-significant-p** check does not happen unless the **significant-p** check succeeds.

Conditionalizing with the Keyword it

In the typical format of a conditionalized clause such as the one below, *expr2* can be the keyword **it**.

when *expr1 keyword expr2*

If that is the case, a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the two clauses below are equivalent:

when *expr* **return it**

thereis *expr*

Similarly you 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 **when**s, **unless**es, and/or **if**s 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.

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*`))`.

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, skipping the epilogue code.

See the section "Conditionalizing with the Keyword **it**".

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.

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.

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))
```

The **return** clause is useful with the conditionalization clauses. It causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is, the two clauses below are equivalent:

when *expr1* **return** *expr2*

when *expr1* **do (return** *expr2***)**

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause iteration to terminate early from being 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))
```

In this case, the **extra-special-significant-p** check does not happen unless the **significant-p** check succeeds.

In the typical format of a conditionalized clause such as the one below, *expr2* can be the keyword **it**.

when *expr1 keyword expr2*

If that is the case, a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the two clauses below are equivalent:

when *expr* **return it**

thereis *expr*

Similarly you 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 **when**s, **unless**es, and/or **if**s 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.

loop Synonyms

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.

This facility exists primarily for diehard users of a predecessor of **loop**. We do not recommend its unconstrained use, as it tends to decrease the portability of your code.

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) ...)
```

This form 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 the function **map-over-properties**, defined 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)))
```

map-over-properties maps *fn* over the properties on *symbol*, giving it arguments of the symbol, the property name, and the value of that property.

The Iteration Framework

Understanding how **loop** constructs iterations is necessary if you are writing your own iteration paths, and can be useful in clarifying what **loop** does with its input.

loop, for the purpose of *stepping*, has 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, that of *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient to describe 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 actually two versions of the sequence described 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 provides first-iteration endtests and steps for the first iteration, the other, the endtests and steps for the remaining iterations. These lists contain dummy entries so that identical expressions appear in the same position in both. When **loop** is done parsing all of the clauses, these lists are merged back together, such that corresponding identical expressions in both lists are not duplicated unless they are simple and it is worth doing.

Thus, you *might* get some duplicated code if you have 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 endtest for any particular iteration can be placed 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 rule, if an iteration-driving clause implicitly supplies an endtest, 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.

Iteration Paths for loop

Iteration paths provide a mechanism for the user to extend 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 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)
```

each can be substituted for the *its* keyword, as can *his*, *her*, or *their*, although this is not common practice, and we do not recommend it. See the section "Predefined Iteration Paths". This section shows some sample uses of iteration paths.

Very often, iteration paths step internal variables that you do not specify, such as an index into some data structure. Although in most cases you may not wish to be concerned with such low-level matters, it is occasionally useful to understand 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)
      ...)
```

This 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". 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.

Predefined Iteration Paths

loop comes with four predefined iteration path functions; one implements a **mapatoms**-like iteration path facility, and another is used for defining iteration paths for stepping through sequences.

The interned-symbols Path

The **interned-symbols** iteration path is like a **mapatoms** for **loop**.

```
(loop for sym being interned-symbols ...)
```

This iterates over all of the symbols in the current package and its superiors. 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 ...)
```

This 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 the code is *unaffected* by changes to the implementation of packages.

Sequence Iteration

One very common form of iteration is 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 you 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 following **loop** clause steps *var* over the elements of *array*, starting from **0**:

```
for var being the array-elements of array
```

The sequence path function also accepts **in** as a synonym for **of**.

The range and stepping of the iteration can be specified with the use of all the same keywords 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 the following form steps *var* over all of the odd elements of *array*:

```
(loop for var being the array-elements of array
          from 1 by 2
          ...)
```

And the following form steps in "reverse" order:

```
(loop for var being the array-elements of array
          downto 0
          ...)
```

The **vector-elements** iteration path can be defined in NIL (which it is) as follows:

```
(define-loop-sequence-path (vector-elements vector-element)
                           vref vector-length notype notype)
```

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 means 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 for **loop**".

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.

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.

`loop` comes with two predefined iteration path functions: the **interned-symbols** path and the **array-elements** path.

The following loop:

```
(loop for sym being interned-symbols ...)
```

iterates over all of the symbols in the current package and its superiors. The particular package to look in can be specified as in:

```
(loop for sym being the interned-symbols in package ...)
```

Also, 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}
          ...)
```

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 **zl:loop** functions, with the effect that it is transparent to changes to the implementation of packages.

One very common form of iteration is that over the elements of some object that is accessible by means of an integer index. **zl: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.

```
(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 **zl: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).

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".) 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**. It 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? Should be **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 to allow a single function to 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 initial-

ization 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".

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 by an iteration path function 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*, and so on, are optional.

A Sample Path Definition

Here is a sample 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)**. Second, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be:

```
(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
```

Using future-common-lisp:loop

What is future-common-lisp:loop?

The macro **future-common-lisp:loop** performs iteration by executing a series of forms one or more times. Loop keywords are symbols recognized by **future-common-lisp:loop**. The provide such capabilites as control of direction of iteration, accumulation of values inside the loop body, and evaluation of expressions that precede or follow the loop body.

For **future-common-lisp:loop** without clauses, each form is evaluated in turn from left to right. When the last form has been evaluated, then the first form is evaluated again, and so on, in a never-ending cycle. **future-common-lisp:loop** establishes an implicit block named **nil**. The execution of **future-common-lisp:loop** can be terminated explicitly, by using **return**, **throw** or **return-from**, for example.

How future-common-lisp:loop Works

Expansion of the **future-common-lisp:loop** macro produces an implicit block named **nil** unless **named** is supplied. Thus, **return** and **return-from** can be used to return values from **future-common-lisp:loop** or to exit **future-common-lisp:loop**. Within the executable parts of loop clauses and around the entire **future-common-lisp:loop** form, variables can be bound by using regular lisp mechanisms, such as **let**.

When Lisp encounters a **future-common-lisp:loop** form, it invokes the loop facility, which expands the loop expression into simpler, less abstract code that implements the loop. The loop facility defines clauses that are introduced by loop keywords. The loop clauses contain forms and loop keywords.

Loop keywords are not true keywords; they are ordinary symbols, recognized by print name, that are meaningful only to the **future-common-lisp:loop** facility. Because loop keywords are recognized by their names, they may be in any package. If no loop keywords are supplied, the loop facility repeatedly executes the loop body.

The **future-common-lisp:loop** macro translates the given form into code and returns the expanded form. The expanded form is one or more lambda expressions for the local binding of loop variables and a **block** and a **tagbody** that express a looping control structure. The variables established in **future-common-lisp:loop** are bound as if by **let** or **lambda**. The assignment of the initial values is always calculated in the order specified by the user. (A variable is sometimes bound to a meaningless value of the correct type, and then later in the prologue it is set to the true initial value by using **setq**.)

After the form is expanded, it consists of three basic parts in the tagbody:

prologue

> The loop prologue contains forms that are executed before iteration begins, such as any automatic variable initializations prescribed by the variable clauses, along with any **initially** clauses in the order they appear in the source.

body

> The loop body contains those forms that are executed during iteration, including application-specific calculations, termination tests, and variable stepping.

epilogue

> The loop epilogue contains forms that are executed after iteration terminates, such as **finally** clauses, if any, along with any implicit return value from an accumulation clause or an end-test clause.

Some clauses from the source form contribute code only to the loop prologue; these clauses must come before other clauses that are in the main body of the **future-common-lisp:loop** form. Others contribute code only to the loop epilogue. All other clauses contribute to the final translated form in the same order given in the original source form of the **future-common-lisp:loop**.

Syntax of future-common-lisp:loop

Notational Conventions and Syntax for future-common-lisp:loop

The syntax for loop constructs is represented as follows:

- Loop keyword names are in **boldface**.
- Symbols are enclosed by braces { }. Braces followed by an asterix (*) indicate that the contents enclosed by the braces can appear any number of times or not

at all.

{*z*}* zero or one occurrences of *z*

A plus sign (+) indicates that the contents enclosed by the braces must appear at least once.

{*z*}+ one or more occurrences of *z*

- Brackets [ ] indicate that the contents enclosed by the brackets is optional and can appear only once.

  [*z*] zero or more occurrences of *z*

- Elements separated by a vertical bar | indicate that either of the elements can appear, but not both.

{**for** | **as**} *var* [*type-spec*] [{**from** | **downfrom** | **upfrom**} *form1*]
[{**to** | **downto** | **upto** | **below** | **above**} *form2*]
[**by** *form3*]

An overview of loop syntax is provided in the following paragraphs. More detailed syntax descriptions of individual clauses are provided in the section "Constructs in **future-common-lisp:loop** ". Syntax for **future-common-lisp:loop** with keyed clauses:

future-common-lisp:loop [named *name*] [*variables*]* [*main*]

variables::= with | *initial-final* | **for** | **as**

initial-final::= initially | finally

main::= *unconditional* | *accumulation* | *conditional* | *end-test* | *initial-final*

unconditional::= do | doing | return

accumulation::= collect | collecting | append | appending
| nconc | nconcing | count | counting | sum | summing
| maximize | maximizing | minimize | minimizing

conditional::= when | if | unless

end-test::= while | until | always | never | thereis | repeat

Syntax for **future-common-lisp:loop** without clauses:

future-common-lisp:loop [*tag* | *form*}*

Syntax for **future-common-lisp:loop-finish**:

(**future-common-lisp:loop-finish**)

Parsing future-common-lisp:loop Clauses

The syntactic parts of the **future-common-lisp:loop** construct are called clauses. The parsing of keywords determines the scope of clause. The following example shows a **future-common-lisp:loop** construct with six clauses:

```
(loop for i from 1 to (compute-top-value)       ; first clause
      while (not (unacceptable i))              ; second clause
      collect (square i)                        ; third clause
      do (format t "Working on ~D now" i)       ; fourth clause
      when (evenp i)                            ; fifth clause
        do (format t "~D is an even number" i)
      finally (format t "About to exit!"))      ; sixth clause
```

Each loop keyword introduces either a compound or a simple loop clause that can consist of a loop keyword followed by a single form. The number of forms in a clause is determined by the loop keyword that begins the clause and by the auxiliary keywords in the clause. The keywords **do**, **initially**, and **finally** are the only loop keywords that can take any number of forms and group them as if in a single **progn** form.

Loop clauses can contain auxiliary keywords, sometimes called prepositions. For example, the first clause in the code above includes the prepositions **from** and **to**, which mark the value from which stepping begins and the value at which stepping ends.

Order of Execution in future-common-lisp:loop

With the exceptions listed below, clauses are executed in the loop body in the order in which they appear in the source. Execution is repeated until a clause terminates the **future-common-lisp:loop** or until a **return**, **go**, or **throw** form is encountered. The following actions are exceptions to the linear order of execution:

- All variables are initialized first, regardless of where the establishing clauses appear in the source. The order of initialization follows the order of these clauses.

- The code for any **initially** clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop prologue after any implicit variable initializations.

- The code for any **finally** clauses is collected into one **progn** in the order in which the clauses appear in the source. The collected code is executed once in the loop epilogue before any implicit values from the accumulation clauses are

returned. Explicit returns anywhere in the source, however, will exit the **future-common-lisp:loop** without executing the epilogue code.

- A **with** clause introduces a variable binding and an optional initial value. The initial values are calculated in the order in which the **with** clauses occur.
- Iteration control clauses implicitly initialize variables, step variables (generally between each execution of the loop body), and perform termination tests (generally just before the execution of the loop body).

Kinds of future-common-lisp:loop Clauses

Loop clauses fall into one of the following six categories:

Variable initialization and stepping

for, as The **for** and **as** constructs provide iteration control clauses that establish a variable to be initialized. **for** and **as** clauses can be combined with the loop keyword **and** to get parallel initialization and stepping. Otherwise, the initialization and stepping are sequential. The **for** and **as** constructs provide a termination test that is determined by the iteration control clause.

with The **with** construct is similar to a single **let** clause. **with** clauses can be combined using the loop keyword **and** to get parallel initialization.

repeat The **repeat** construct causes iteration to terminate after a specified number of times. It uses an internal variable to keep track of the number of iterations.

Value accumulation

collect The **collect** construct takes one *form* in its clause and adds the value of that *form* to the end of a list of values. By default, the list of values is returned when the **future-common-lisp:loop** finishes.

```
;; Collect all the symbols in a list.
 (loop for i in '(bird 3 4 turtle (1 . 4) horse cat)
       when (symbolp i) collect i)
=> (BIRD TURTLE HORSE CAT)

;; Collect and return odd numbers.
 (loop for i from 1 to 10
       if (oddp i) collect i)
=> (1 3 5 7 9)

;; Collect items into local variable,
;; but don't return them.
 (loop for i in '(a b c d) by #'cddr
       collect i into my-list
       finally (print my-list))
(A C)
=> NIL
```

append The **append** construct takes one *form* in its clause and appends the value of that *form* to the end of a list of values. By default, the list of values is returned when the **future-common-lisp:loop** finishes.

```
;; Use APPEND to concatenate some sublists.
  (loop for x in '((a) (b) ((c)))
        append x)
=> (A B (C))
```

nconc The **nconc** construct is similar to the **append** construct, but its **list** values are concatenated as if by the function **nconc**. By default, the **list** of values is returned when the **future-common-lisp:loop** finishes.

```
;; NCONC some sublists together.  Note that only lists
;; made by the call to LIST are modified.
  (loop for i upfrom 0
        as x in '(a b (c))
        nconc (if (evenp i) (list x) nil))
=> (A (C))
```

sum The **sum** construct takes one *form* in its clause that must evaluate to a **number** and accumulates the sum of all these **numbers**. By default, the cumulative sum is returned when the **future-common-lisp:loop** finishes.

```
(loop for i fixnum in '(1 2 3 4 5)
       sum i)
=> 15
 (setq series '(1.2 4.3 5.7))
=> (1.2 4.3 5.7)
 (loop for v in series
       sum (* 2.0 v))
=> 22.4
```

count

The **count** construct takes one *form* in its clause and counts the number of times that the *form* evaluates to a non-**nil** value. By default, the count is returned when the **future-common-lisp:loop** finishes.

```
 (loop for i in '(a b nil c nil d e)
       count i)
=> 5
```

minimize

The **minimize** construct takes one *form* in its clause and determines the minimum value obtained by evaluating that *form*. By default, the minimum value is returned when the **future-common-lisp:loop** finishes.

```
(loop for i in '(2 1 5 3 4)
       minimize i)
=> 1
;; In this example, FIXNUM applies to the variable RESULT.
(loop for v float in series
        minimize (round v) into result fixnum
        finally (return result))
=> 1
(loop for i in '(2 1 5 3 4)
        minimize i)
=> 1
```

maximize

The **maximize** construct takes one *form* in its clause and determines the maximum value obtained by evaluating that *form*. By default, the maximum value is returned when the **future-common-lisp:loop** finishes.

```
 (loop for i in '(2 1 5 3 4)
       maximize i)
 => 5

;; In this example, FIXNUM applies to the internal
;; variable that holds the maximum value.
 (setq series '(1.2 4.3 5.7))
=> (1.2 4.3 5.7)
 (loop for v in series
       maximize (round v) fixnum)
=> 6
```

Termination conditions

while
The **while** construct takes one *form*, a **condition**, and terminates the iteration if the **condition** evaluates to **nil**. A **while** clause is equivalent to the expression (**if** (not **condition**) (**future-common-lisp:loop-finish**)).

until
The **until** construct is the inverse of **while**; it terminates the iteration if the condition evaluates to any non-**nil** value. An **until** clause is equivalent to the expression (**if condition** (**future-common-lisp:loop-finish**)).

always
The **always** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**.

never
The **never** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **non-nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**.

thereis
The **thereis** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **non-nil**; in this case, it returns that value.

future-common:loop-finish
The **future-common-lisp:loop-finish** macro terminates iteration and returns any accumulated result. Any **finally** clauses that are supplied are evaluated.

Unconditional execution

do
The **do** construct evaluates all forms in its clause.

return
The **return** construct takes one form and returns its value. It is equivalent to the clause **do** (**return it value**).

```
;; Print numbers and their squares.
;; The DO construct applies to multiple forms.
 (loop for i from 1 to 3
       do (print i)
          (print (* i i)))
1
1
2
4
3
9
=> NIL
```

Conditional execution

if

The **if** construct takes one *form* as a predicate and a clause that is executed when the predicate is true. The clause can be a value accumulation, unconditional, or another conditional clause; it can also be any combination of such clauses connected by the loop keyword **and**.

when

The **when** construct is a synonym for the **if** construct.

```
;; Signal an exceptional condition.
 (loop for item in '(1 2 3 a 4 5)
       when (not (numberp item))
        return (cerror "Enter a new value" "Non-numeric value: ~s" item))
Error: Non-numeric value: A

;; The previous example is equivalent to the following one.
 (loop for item in '(1 2 3 a 4 5)
       when (not (numberp item))
        do (return
            (cerror "Enter a new value" "Non-numeric value: ~s" item)))
Error: Non-numeric value: A

;; This example parses a simple printed string representation from
;; BUFFER (which is itself a string) and returns the index of the
;; closing double-quote character.
(let ((buffer "\"foo\" \"bar\""))
  (loop initially (unless (char= (char buffer 0) #\")
                    (loop-finish))
        for i fixnum from 1 below (string-length buffer)
        when (char= (char buffer i) #\")
         return i))
=> 4

;; The FINALLY clause prints the last value of I.
;; The collected value is returned.
 (loop for i from 1 to 10
       when (> i 5)
         collect i
       finally (print i))
=> 4
(6 7 8 9 10)
```

```
;; Return both the count of collected numbers and the numbers.
 (loop for i from 1 to 10
       when (> i 5)
         collect i into number-list
         and count i into number-count
       finally (return (values number-count number-list)))
=> 5
(6 7 8 9 10)
```

unless The **unless** construct is similar to **when** except that it complements the predicate.

else The **else** construct provides an optional component of **if**, **when**, and **unless** clauses that is executed when the predicate is false. The component is one of the clauses described under **if**.

end The **end** construct provides an optional component to mark the end of a conditional clause.

Miscellaneous operations

named The **named** construct gives a name for the block of the loop.

```
;; Just name and return.
 (loop named max
       for i from 1 to 10
       do (print i)
       do (return-from max 'done))
1
=> DONE
```

initially The **initially** construct causes its forms to be evaluated in the loop prologue, which precedes all **future-common-lisp:loop** code except for initial settings supplied by the constructs **with**, **for**, or **as**.

finally The **finally** construct causes its forms to be evaluated in the loop epilogue after normal iteration terminates. An unconditional clause can also follow the loop keyword **finally**.

Constructs in future-common-lisp:loop

This section describes the constructs provided by **future-common-lisp:loop**. The constructs are grouped according to function into the following categories:

- Iteration Control

- End-Test Control

- Value Accumulation
- Variable Initializations
- Conditional Execution
- Unconditional Execution
- Miscellaneous Features

Iteration Control in future-common-lisp:loop

Iteration control clauses allow direction of **future-common-lisp:loop** iteration. The loop keywords **for**, **as**, and **repeat** designate iteration control clauses. Iteration control clauses differ with respect to the specification of termination conditions and to the initialization and stepping of loop variables. Iteration clauses by themselves do not cause the loop facility to return values, but they can be used in conjunction with value-accumulation clauses to return values.

All variables are initialized in the loop prologue. The scope of the variable binding is lexical unless it is proclaimed **special**; thus, the variable can be accessed only by forms that lie textually within the **future-common-lisp:loop**. Stepping assignments are made in the loop body before any other forms are evaluated in the body.

The variable argument in iteration control clauses can be a destructuring list. A destructuring list is a tree whose non-**null** atoms are **symbols** that can be assigned a value. See the section "Destructuring in **future-common-lisp:loop**".

The iteration control clauses **for**, **as**, and **repeat** must precede any other loop clauses, except **initially**, **with**, and **named**, since they establish variable bindings. When iteration control clauses are used in a **future-common-lisp:loop**, termination tests in the loop body are evaluated before any other loop body code is executed.

If multiple iteration clauses are used to control iteration, variable initialization and stepping occur sequentially by default. The **and** construct can be used to connect two or more iteration clauses when sequential binding and stepping are not necessary. The iteration behavior of clauses joined by **and** is analogous to the behavior of the macro **do** with respect to **do***.

for and as Constructs

The **for** and **as** clauses iterate by using one or more local loop variables that are initialized to some value and that can be modified or stepped after each iteration. For these clauses, iteration terminates when a local variable reaches some supplied value or when some other loop clause terminates iteration. At each iteration, variables can be **word** stepped by an increment or a decrement or can be assigned a new value by the evaluation of a form. Destructuring can be used to assign initial values to variables during iteration.

The **for** and **as** keywords are synonyms; they can be used interchangeably. There are seven syntactic formats for these constructs. In each syntactic format, the type of *var* can be supplied by the optional *type-spec* argument. If *var* is a destructuring list, the type supplied by the *type-spec* argument must appropriately match the elements of the list.

Syntax 1:

{**for** | **as**} *var* [*type-spec*] [{**from** | **downfrom** | **upfrom**} *form1*]
[{**to** | **downto** | **upto** | **below** | **above**} *form2*] [**by** *form3*]

The **for** or **as** construct iterates from the value supplied by *form1* to the value supplied by *form2* in increments or decrements denoted by *form3*. Each expression is evaluated only once and must evaluate to a number.

The variable *var* is bound to the value of *form1* in the first iteration and is stepped by the value of *form3* in each succeeding iteration, or by 1 if *form3* is not provided.

The following loop keywords serve as valid prepositions within this syntax, and at least one must be used in any **for** or **as** construct:

from *form1*
The loop keyword **from** marks the value from which stepping begins, as supplied by *form1*. Stepping is incremental by default. If decremental stepping is desired, the preposition **downto** or **above** must be used with *form2*. For incremental stepping, the default **from** value is 0.

downfrom, **upfrom** *form1*
The loop keyword **downfrom** indicates that the variable *var* is decreased in decrements supplied by *form3*; the loop keyword **upfrom** indicates that *var* is increased in increments supplied by *form3*.

to *form2*
The loop keyword **to** marks the end value for stepping supplied in *form2*. Stepping is incremental by default. If decremental stepping is desired, the preposition **downto**, **downfrom**, or **above** must be used with *form2*.

downto, **upto** *form2*
The loop keyword **downto** allows iteration to proceed from a larger number to a smaller number by the decrement *form3*. The loop keyword **upto** allows iteration to proceed from a smaller number to a larger number by the increment *form3*. Since there is no default for *form1* in decremental stepping, a value must be supplied with **downto**.

below, **above** *form2*
The loop keywords **below** and **above** are analogous to **upto** and **downto** respectively. These keywords stop iteration just before the value of the variable *var* reaches the value supplied by *form2*; the end value of *form2* is not included. Since there is

no default for *form1* in decremental *stepping*, a value must be supplied with **above**.

by *form3* — The loop keyword **by** marks the increment or decrement supplied by *form3*. The value of *form3* can be any positive *number*. The default value is 1.

In an iteration control clause, the **for** or **as** construct causes termination when the supplied limit is reached. That is, iteration continues until the value *var* is stepped to the exclusive or inclusive limit supplied by *form2*. The range is exclusive if *form3* increases or decreases *var* to the value of *form2* without reaching that value; the loop keywords **below** and **above** provide exclusive limits. An inclusive limit allows *var* to attain the value of *form2*; **to**, **downto**, and **upto** provide inclusive limits.

By convention, **for** introduces new iterations and **as** introduces iterations that depend on a previous iteration specification.

```
;; Print some numbers.
 (loop as i from 1 to 3
       do (print i))
1
2
3
=> NIL

;; Print every third number.
 (loop for i from 10 downto 1 by 3
       do (print i))
10
7
4
1
=> NIL

;; Step incrementally from the default starting value.
 (loop as i below 3
       do (print i))
0
1
2
=> NIL
```

Syntax 2:

{**for** | **as**} *var* [*type-spec*] **in** *form1* [**by** *step-fun*]

The **for** or **as** construct iterates over the contents of a list. It checks for the end of the list as if by using **endp**. The variable *var* is bound to the successive elements of the list in *form1* before each iteration. At the end of each iteration, the

function *step-fun* is applied to then list; the default value for *step-fun* is **cdr**. The loop keywords **in** and **by** serve as valid prepositions in this syntax. The **for** or **as** construct causes termination when the end of the list is reached. For example:

```
;; Print every item in a list.
 (loop for item in '(1 2 3) do (print item))
1
2
3
=> NIL

;; Print every other item in a list.
 (loop for item in '(1 2 3 4 5) by #'cddr
       do (print item))
1
3
5
=> NIL

;; Destructure a list, and sum the x values using fixnum arithmetic.
(loop for (item . x) (t . fixnum) in '((A . 1) (B . 2) (C . 3))
       unless (eq item 'B) sum x)
=> 4
```

Syntax 3:

{**for** | **as**} *var* [*type-spec*] **on** *form1* [**by** *step-fun*]

The **for** or **as** construct iterates over the contents of a list. It checks for the end of the list as if by using **endp**. The variable *var* is bound to the successive tails of the *list* in *form1*. At the end of each iteration, the function *step-fun* is applied to the list; the default value for *step-fun* is **cdr**. The loop keywords **on** and **by** serve as valid prepositions in this syntax. The **for** or **as** construct causes termination when the end of the list is reached. The following example demonstrates the **for-as-on-list** subclause:

```
;; Collect successive tails of a list.
 (loop for sublist on '(a b c d)
       collect sublist)
=>((A B C D) (B C D) (C D) (D))

;; Print a list by using destructuring with the loop keyword ON.
 (loop for (item) on '(1 2 3)
       do (print item))
1
2
3
=> NIL
```

Syntax 4:

{**for** | **as**} *var* [*type-spec*] = *form1* [**then** *form2*]

The **for** or **as** construct initializes the variable *var* by setting it to the result of evaluating *form1* on the first iteration, then setting it to the result of evaluating *form2* on the second and subsequent iterations. If *form2* is omitted, the construct uses *form1* on the second and subsequent iterations. The loop keywords = and **then** serve as valid prepositions in this syntax. This construct does not provide any termination conditions. For example:

```
;; Collect some numbers.
 (loop for item = 1 then (+ item 10)
       for iteration from 1 to 5
       collect item)
=> (1 11 21 31 41)
```

Syntax 5:

{**for** | **as**} *var* [*type-spec*] **across** *vector*

The **for** or **as** construct binds the variable *var* to the value of each element in the array *vector*. The loop keyword **across** marks the array *vector*; **across** is used as a preposition in this syntax. Iteration stops when there are no more elements in the supplied array that can be referenced. Some implementations might recognize a **the** special form in the *vector* form to produce more efficient code. For example:

```
(loop initially (terpri) "foo"
       do (write-char char stream))
foo
=> NIL
```

Syntax 6:

{**for** | **as**} *var* [*type-spec*] **being** {**each** | **the**}
{**hash-key**[**s**] | **hash-value**[**s**]}
{**in** | **of**} *hash-table* [**using** ({**hash-key** | **hash-value**} *other-var*)]

The **for** or **as** construct iterates over the elements, keys, and values of a hash table. In this syntax, a compound preposition is used to designate access to a hash table. The variable *var* takes on the value of each hash key or hash value in the supplied hash table. The following loop keywords serve as valid prepositions within this syntax:

being — The keyword **being** introduces either the loop method **hash-key** or hash-**value**.

each, **the** — The loop keyword **each** follows the loop keyword **being** when **hash-key** or **hash-value** is used. The loop keyword **the** is used with **hash-keys** and **hash-values** only for ease of reading. This agreement isn't required.

hash-key, **hash-keys**
These loop keywords access each key entry of the *hash-table*. If the name *hash-value* is supplied in a **using** construct with one of these loop methods, the iteration can optionally access the keyed value. The order in which the keys are accessed is undefined; empty slots in the *hash-table* are ignored.

hash-value, **hash-values**
These loop keywords access each value entry of a **hash-table**. If the name **hash-key** is supplied in a **using** construct with one of these loop methods, the iteration can optionally access the key that corresponds to the value. The order in which the keys are accessed is undefined; empty slots in the **hash-table** are ignored.

using The loop keyword **using** introduces the optional key or the keyed value to be accessed. It allows access to the hash key if iteration is over the hash values, and the hash value if iteration is over the hash keys

in, **of** These loop prepositions introduce *hash-table*.

In effect the following expression is a compound preposition:

being [**each** | **the**] [**hash-value** | **hash-values** | **hash-key** | **hash-key**] [**in** | **of**]

Iteration stops when there are no more hash keys or hash values to be referenced in the supplied hash table.

Syntax 7:

{**for** | **as**} *var* [*type-spec*] **being** {**each** | **the**}
{**symbol**[**s**] | **present-symbol**[**s**] | **external-symbol**[**s**]}
[{**in** | **of**} *package*]

The **for** or **as** construct iterates over the symbols in a package. In this syntax, a compound preposition is used to designate access to a package. The variable *var* takes on the value of each symbol in the supplied package. The following loop keywords serve as valid prepositions within this syntax:

being The keyword **being** introduces either the loop method **symbol**[**s**], **present-symbol**[**s**], or **external-symbol**[**s**].

each, **the** The loop keyword **each** follows the loop keyword **being** when **symbol**, **present-symbol**, or **external-symbol** is used. The loop keyword **the** is used with **symbols**, **present-symbols**, and **external-symbols** only for ease of reading. This agreement isn't required.

present-symbol, **present-symbols**
These loop methods iterate over the symbols that are present but not external in a given *package*. The package to be iterated over is supplied in the same way that package arguments to **find-package** are supplied. If the package for the iteration is not supplied, the current package is used. If a package that does not exist is supplied, an error of type **package-error** is signalled.

symbol, **symbols** These loop methods iterate over **symbols** that are accessible from a given *package*. The package to be iterated over is supplied in the same way package arguments to **find-package** are supplied. If the package for the iteration is not supplied, the current package is used. If a package that does not exist is supplied, an error of type **conditions:package-error** is signalled.

external-symbol, **external-symbols**
These loop methods iterate over the external **symbols** of the given *package*. The package to be iterated over is supplied in the same way package arguments to **find-package** are supplied. If the package for the iteration is not supplied, the current package is used. If a package that does not exist is supplied, an error of type **package-error** is signalled.

in, **of** These loop prepositions mark the package *package*.

In effect

[**being**] [**each** | **the**] [[[**present** | **external**] **symbol**] |
[[**present** | **external**] **symbols**]] [**in** | **of**]

is a compound preposition. Iteration stops when there are no more symbols to be referenced in the supplied *package*.

The repeat Construct in future-common-lisp:loop

repeat form

The **repeat** construct causes iteration to terminate after a specified number of times. The loop body executes *n* times, where *n* is the value of the expression *form*. The *form* argument is evaluated once in the loop prologue. If the expression evaluates to 0 or to a negative number, the loop body is not evaluated. The following example demonstrates the **repeat** construct:

```
(loop repeat 3
      do (format t "~&What I say three times is true.~%"))
   What I say three times is true
   What I say three times is true
   What I say three times is true
=> NIL
```

```
(loop repeat -15
   do (format t "~&What you see is what you expect.~%"))
=> NIL
```

End-Test Control in future-common-lisp:loop

The following loop keywords designate constructs that use a single test condition to determine when loop iteration should terminate:

always, **never**, **thereis**

while, **until**

The constructs **always**, **never**, and **thereis** provide specific values to be returned when a looop terminates. Using **always**, **never**, or **thereis** in a loop with value-returning accumulation clauses that are not **into** causes an error of type **conditions:program-error** to be signalled. Since **always**, **never**, and **thereis** use the macro **return** to terminate iteration, any **finally** clause that is supplied is not evaluated. In all other respects these constructs behave like the **while** and **until** constructs.

The macro **future-common-lisp:loop-finish** can be used at any time to cause regular termination. In regular termination, **finally** clauses are executed and default return values are returned.

always, never, thereis

always *form* The **always** construct takes one *form* and terminates the **future-common-lisp:loop** if the *form* ever evaluates to **nil**; in this case, it returns **nil**. Otherwise, it provides a default return value of **t**. If the value of the supplied *form* is never **nil**, some other construct can terminate the iteration. Otherwise, it provides a default return value of **t**.

```
;; Make sure I is always less than 11 (two ways).
;; The FOR construct terminates these loops.
 (loop for i from 0 to 10
       always (< i 11))
=> T
```

never *form*

The **never** construct terminates iteration the first time that the value of the supplied **form** is non-**nil**; the **future-common-lisp:loop** returns **nil**. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **t**.

```
(loop for i from 0 to 10
      never (> i 11))
=>T
```

thereis *form*

The **thereis** construct takes one **form** and terminates the **loop** if the *form* ever evaluates to non-**nil**; in this case, it returns that value. The **thereis** construct terminates iteration the first time that the value of the supplied *form* is non-**nil**; the **future-common-lisp:loop** returns the value of the supplied *form*. If the value of the supplied *form* is always **nil**, some other construct can terminate the iteration. Unless some other clause contributes a return value, the default value returned is **nil**.

```
;; If I exceeds 10 return I; otherwise, return NIL.
;; The THEREIS construct terminates this loop.
 (loop for i from 0
      thereis (when (> i 10) i) ) => 11
```

There are two differences between the **thereis** and **until** constructs:

- The **until** construct does not contribute a return value based on the value of the supplied *form*.

- The **until** construct executes any **finally** clause. Since **thereis** uses the macro **return** to terminate iteration, any **finally** clause that is supplied is not evaluated.

```
;;; The FINALLY clause is not evaluated in these examples.
 (loop for i from 0 to 10
       always (< i 9)
       finally (print "you won't see this"))
=> NIL
 (loop never t
       finally (print "you won't see this"))
=> NIL
 (loop thereis "Here is my value"
       finally (print "you won't see this"))
=> "Here is my value"

;; The FOR construct terminates this loop, so the FINALLY clause
;; is evaluated.
 (loop for i from 1 to 10
       thereis (> i 11)
       finally (print i))
   11
=> NIL

;; If this code could be used to find a counterexample to Fermat's
;; last theorem, it would still not return the value of the
;; counterexample because all of the THEREIS clauses in this example
;; only return T.  Of course, this code does not terminate.

 (loop for z upfrom 2
       thereis
         (loop for n upfrom 3 below (log z 2)
               thereis
                 (loop for x below z
                       thereis
                         (loop for y below z
                               thereis (= (+ (expt x n) (expt y n))
                                          (expt z n))))))

; The finally clause is not evaluated.

 (loop never t
       finally (print "You won't see this."))
=> NIL
```

while, until

while *form* — The **while** construct allows iteration to continue until the supplied *form* evaluates to **nil**. The supplied *form* is reevaluated at the location of the **while** clause.

```
(loop while (hungry-p) do (eat))

;; UNTIL (NOT...) is equivalent to WHILE.
 (loop until (not (hungry-p)) do (eat))

;; Collect the length and the items of STACK.
 (let ((stack '(a b c d e f)))
   (loop while stack
         for item = (length stack) then (pop stack)
         collect item))
=> (6 A B C D E F)
```

until *form*

The **until** construct is equivalent to **while** (**not** *form*) dots. If the value of the supplied *form* is non-**nil**, iteration terminates.

```
;; Use WHILE to terminate a loop that otherwise
;; wouldn't terminate.
;; Note that WHILE occurs after the WHEN.
 (loop for i fixnum from 3
       when (oddp i) collect i
       while (< i 5))
=> (3 5)
```

The **while** and **until** constructs can be used at any point in a **future-common-lisp:loop**. If an **until** or **while** clause causes termination, any clauses that precede it in the source are still evaluated. If the **until** and **while** constructs cause termination, control is passed to the loop epilogue, where any **finally** clauses will be executed.

There are two differences between the **never** and **until** constructs:

- The **until** construct does not contribute a return value based on the value of the supplied *form*.

- The **until** construct executes a **finally** clause. Since **never** uses the macro **return** to terminate iteration, any **finally** clause that is supplied is not evaluated.

In most cases it is not necessary to use **future-common-lisp:loop-finish** because other loop control clauses terminate the **future-common-lisp:loop**. The macro **future-common-lisp:loop-finish** is used to provide a normal exit from a nested condition inside a **future-common-lisp:loop**.

In normal termination, **finally** clauses are executed and default return values are returned. Since **future-common-lisp:loop-finish** transfers control to the loop epilogue, using **future-common-lisp:loop-finish** within a **finally** expression can cause infinite looping. It is implementation dependent whether or not, in a particular

future-common-lisp:loop invocation, **future-common-lisp:loop-finish** is a global *macro* or a local one (created as if by **macrolet**).

End-test control constructs can be used anywhere within the loop body. The termination conditions are tested in the order in which they appear.

Value Accumulation in future-common-lisp:loop

Accumulating values during iteration and returning them from a loop is often useful. Some of these accumulations occur so frequently that special loop clauses have been developed to handle them.

The following loop keywords designate clauses that accumulate values in lists and return them:

- **append**, **appending**
- **collect**, **collecting**
- **nconc**, **nconcing**

The following loop keywords designate clauses that accumulate and return numerical values:

- **count**, **counting**
- **maximize**, **maximizing**
- **minimize**, **minimizing**
- **sum**, **summing**

Value-returning accumulation clauses can be combined in a **loop** if all the clauses accumulate the same type of object. By default, the loop facility returns only one value; thus, the objects collected by multiple accumulation clauses as return values must have compatible types. For example, since both the **collect** and **append** constructs accumulate objects into a list that is returned from a **future-common-lisp:loop**, they can be combined safely.

```
;; Collect every name and the kids in one list by using
;; COLLECT and APPEND.
 (loop for name in '(fred sue alice joe june)
       for kids in '((bob ken) () () (kris sunshine) ())
       collect name
       append kids)
=> (FRED BOB KEN SUE ALICE JOE KRIS SUNSHINE JUNE)
```

Multiple clauses that do not accumulate the same type of object can coexist in a **future-common-lisp:loop** only if each clause accumulates its values into a different user-specified variable.

collect, collecting

collect[**ing**] *form* During each iteration, the constructs **collect** and **collecting** collect the value of the supplied form into a list. When iteration terminates, the list is returned. The argument *var* is set to the list of collected values; if *var* is supplied, the **future-common-lisp:loop** does not return the final list automatically. If *var* is not supplied, it is equivalent to supplying an internal name for *var* and returning its value in a **finally** clause. The *var* argument is bound as if by the construct **with**. A type cannot be supplied for *var*; it must be of type list.

append, appending, nconc, nconcing

append[**ing**] *form* [**into** *var*], **nconc**[**ing**] *form* [**into** *var*]

The constructs **append**, **appending**, **nconc**, and **nconcing** are similar to **collect** except that the values of the supplied form must be lists.

- The **append** keyword causes its list values to be concatenated into a single list, as if they were arguments to the function **append**.

- The **nconc** keyword causes its list values to be concatenated into a single list, as if they were arguments to the function **nconc**.

The argument *var* is set to the list of concatenated values; if *var* is supplied, **future-common-lisp:loop** does not return the final list automatically. The *var* argument is bound as if by the construct **with**. A type cannot be supplied for *var*; it must be of type **list**. The construct **nconc** destructively modifies its argument lists. The **append** construct is similar to **collect** except the values of the supplied form must be lists. These lists are not modified but are concatenated together into a single list, as if they were arguments to **append**. The argument *var* is bound to the list of concatenated values; if *var* is supplied, the loop does not return the final list automatically. The *var* argument is bound as if by the construct **with**. A type cannot be supplied for *var*; it must be of type **list**.

count, counting

count[**ing**] *form* [**into** *var*] [*type-spec*]

The **count** construct counts the number of times that the supplied *form* has a non-**nil** value. The argument *var* accumulates the number of occurrences; if *var* is supplied, **future-common-lisp:loop** does not return the final count automatically. The *var* argument is bound as if by the construct **with**. If **into** *var* is

used, a type can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric *type* is supplied. If there is no **into** variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default type is implementation-dependent; but it must be a *subtype* of (or integer **float**).

maximize, maximizing, minimize, minimizing

maximize | **maximizing** *form* [**into** *var*] [*type-spec*]

The **maximize** construct compares the value of the supplied form obtained during the first iteration with values obtained in successive iterations. The maximum value encountered is determined and returned. If **future-common-lisp:loop** never executes the body, the returned value is unspecified. The argument *var* accumulates the maximum or minimum value; if *var* is supplied, **future-common-lisp:loop** does not return the maximum or minimum automatically. The *var* argument is bound as if by the construct **with**. If **into** *var* is used, a type can be supplied for *var* with the *type-spec* argument; the consequences are unspecified if a nonnumeric type is supplied. If there is no **into** variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default type must be a subtype of (or integer **float**).

minimize | **minimizing** *form* [**into** *var*] [*type-spec*]

The **minimize** construct is similar to **maximize**; it determines and returns the minimum value. The **minimize** construct compares the value of the supplied *form* obtained during the first iteration with values obtained in successive iterations. The minimum value encountered is determined and returned. If **future-common-lisp:loop** never iterates, the returned value is not meaningful. The argument *var* is bound to the minimum value; if *var* is supplied, the **future-common-lisp:loop** does not return the minimum automatically. The *var* argument is bound as if by the construct **with**. The *type-spec* argument supplies the type for *var*; the default type is **fixnum**. The consequences are unspecified if a nonnumeric type is supplied for *var*.

sum, summing

sum[**ing**] *form* [**into** *var*] [*type-spec*]

The **sum** construct forms a cumulative sum of the values of the supplied *form* at each iteration. The argument *var* is used to accumulate the sum; if *var* is supplied, **future-common-lisp:loop** does not return the final sum automatically. The *var* argument is bound as if by the construct **with**. If **into** *var* is used, a type can be supplied for *var* with the *type-spec* argu-

ment; the consequences are unspecified if a nonnumeric type is supplied. If there is no **into** variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. The default type must be a subtype of **number**.

The loop preposition **into** can be used to name the variable used to hold partial accumulations. The variable is bound as if by the loop construct **with**. If **into** is used, the construct does not provide a default return value; however, the variable is available for use in any **finally** clause.

Local Variable Initializations in future-common-lisp:loop

At the time when the loop facility is invoked, the local variables are bound and are initialized to some value. These local variables exist until **future-common-lisp:loop** iteration terminates, at which point they cease to exist. Implicitly, variables are also established by iteration control clauses and the **into** preposition of accumulation.

with

with *var1* [*type-spec*] [= *form1*] [**and** *var2* [*type-spec*] [= *form2*]

The **with** construct initializes variables that are local to a loop. The variables are initialized one time only. If the optional *type-spec* argument is supplied for the variable *var*, but there is no related expression to be evaluated, *var* is initialized to an appropriate default value for its type. For example, for the types **t**, **number**, and **float**, the default values are **nil**, **0**, and **0.0** respectively. The consequences are unspecified if a *type-spec* argument is supplied for *var* if the related expression returns a value that is not of the supplied type.

Sequential and Parallel Initialization

By default, the **with** construct initializes variables sequentially; that is, one variable is assigned a value before the next expression is evaluated. However, by using the loop keyword **and** to join several **with** clauses, initializations can be forced to occur in parallel; that is, all of the supplied forms are evaluated, and the results are bound to the respective variables simultaneously.

Sequential binding is used when it is desirable for the initialization of some variables to depend on the values of previously bound variables. For example, suppose the variables **a**, **b**, and **c** are to be bound in sequence:

```
;; These bindings occur in sequence.
 (loop with a = 1
       with b = (+ a 2)
       with c = (+ b 3)
       return (list a b c))
=> (1 3 6)


;; These bindings occur in parallel.
 (setq a 5 b 10)
=> 10
 (loop with a = 1
       and b = (+ a 2)
       and c = (+ b 3)
       return (list a b c))
=> (1 7 13)
```

The execution of the previous example of **future-common-lisp:loop** is equivalent to the execution of the following code:

```
 (let* ((a 1)
        (b (+ a 2))
        (c (+ b 3)))
   (block nil
     (tagbody
         (next-loop (return (list a b c))
                    (go next-loop)
                    end-loop))))
;; This example shows a shorthand way to declare local variables
;; that are of different types.
 (loop with (a b c) (float integer float)
       return (format nil "~A ~A ~A" a b c))
=> "0.0 0 0.0"

;; This example shows a shorthand way to declare local variables
;; that are the same type.
 (loop with (a b c) float
       return (format nil "~A ~A ~A" a b c))
=> "0.0 0.0 0.0"
```

If the values of previously bound variables are not needed for the initialization of other local variables, an **and** clause can be used to force the bindings to occur in parallel:

```
(loop with a = 1
       and b = 2
       and c = 3
       return (list a b c))
(1 2 3)
```

The execution of the above loop is equivalent to the execution of the following code:

```
(let ((a 1)
      (b 2)
      (c 3))
  (block nil
    (tagbody
        (next-loop (return (list a b c))
                   (go next-loop)
                   end-loop))))
```

Conditional Execution in future-common-lisp:loop

If the supplied condition is true, the succeeding loop clause is executed. If the supplied condition is not true, the succeeding clause is skipped, and program control moves to the clause that follows the loop keyword **else**. If the supplied condition is not true and no **else** clause is supplied, control is transferred to the clause or construct following the supplied condition. The following keywords designate constructs that are useful when you want loop clauses to operate under a specified condition:

when, if, unless

{**if** | **when** | **unless**} *form clause1* [**and** *clause*]* [**end**]

{**if** | **when** | **unless**} *form clause1* [**and** *clause*]*
else *clause2* [**and** *clause*]* [**end**]

The constructs **if** and **when** allow execution of loop clauses conditionally. These constructs are synonyms and can be used interchangeably. If the value of the test expression *form* is non-**nil**, the expression *clause1* is evaluated. If the test expression evaluates to **nil** and an **elsne** construct is supplied, the statements that follow the **else** are evaluated; otherwise, control passes to the next clause. If **if** or **when** clauses are nested, each **else** is paired with the closest preceding **if** or **when** construct that has no associated **else**.

The **unless** construct is equivalent to **when (not** *form***)** and **if (not** *form***)**. If the value of the test expression form is **nil**, the expression **clause1** is evaluated. If the test expression evaluates to **non-nil** and an **else** construct is supplied, the statements that follow the **else** are evaluated; otherwise, no conditional statement is evaluated. The clause arguments must be either accumulation, unconditional, or conditional clauses.

Clauses that follow the test expression can be grouped by using the loop keyword **and** to produce a conditional block consisting of a compound clause.

The loop keyword **it** can be used to refer to the result of the test expression in a clause. If multiple clauses are connected with **and**, the **it** construct must be the first clause in the block. Since **it** is a loop keyword, **it** cannot be used as a local variable within **future-common-lisp:loop**.

The optional loop keyword **end** marks the end of the clause. If this keyword is not supplied, the next loop keyword marks the end. The construct **end** can be used to distinguish the scoping of compound clauses.

Unconditional Execution in future-common-lisp:loop

The following loop construct evaluates its specified expression wherever it occurs in the expanded form of loop:

do, **doing**

The following loop construct takes one form and returns its value. It is equivalent to the clause (**do** (**return** *value*)).

return

do, doing

do[**ing**] [*form*]*

The *form* argument can be an nonatomic Common Lisp form Each *form* is evaluated in every iteration. The constructs **do**, **initially**, and **finally** are the only loop keywords that take an arbitrary number of forms and group them as if by using an implicity **progn**.

Miscellaneous Features in future-common-lisp:loop

future-common-lisp:loop provides the **name** construct to name a loop so that special form **return-from** can be used.

The loop keywords **initially** and **finally** designate loop constructs that cause expressions to be evaluated before and after the loop body, respectively.

The code for any **initially** clauses is collected into one **progn** in the order in which the clauses appeared in the loop. The collected code is executed once in the loop prologue after any implicit variable initializations.

The code for any **finally** clauses is collected into one **progn** in the order in which the clauses appeared in the loop. The collected code is executed once in the loop epilogue before any implicit values are returned from the accumulation clauses. Explicit returns in the loop body, however, will exit the loop without executing the epilogue code.

Data Types in future-common-lisp:loop

Many loop constructs take a *type-spec* argument that allows you to specify certain data types for loop variables. While it is not necessary to specify a data type for any variable, by doing so you ensure that the variable has a correctly typed initial value. The type declaration is made available to the compiler for more efficient **future-common-lisp:loop** expansion. The *type-spec* argument has the following syntax:

type-spec::= **of-type** *d-type-spec*

d-type-spec::= *type-specifier* | *d-type-spec* . *d-type-spec*

The *type-specifier* argument can be any Common Lisp type specifier. The *d-type-spec* argument is used for destructuring, as described in the section "Destructuring in **future-common-lisp:loop**". If the *d-type-spec* argument consists solely of the types **fixnum**, **float**, **t**, or **nil**, the **of-type** is optional.

The **of-type** construct is optional in these cases to provide backwards compatibility; thus, the following two expressions are the same:

```
;;; This expression uses the old syntax for type specifiers.
 (loop for i fixnum upfrom 3 ...)

;;; This expression uses the new syntax for type specifiers.
 (loop for i of-type fixnum upfrom 3 ...)

;; Declare X and Y to be of type VECTOR and FIXNUM respectively.
 (loop for (x y) of-type (vector fixnum)
       in l do ...)
```

Destructuring in future-common-lisp:loop

Destructuring allows binding of a set of variables to a corresponding set of values anywhere that a value can normally be bound to a single variable. During **future-common-lisp:loop** expansion, each variable in the variable list is matched with the values in the values list. If there are more variables in the variable list than there are values in the values list, the remaining variables are given a value of **nil**. If there are more values than variables listed, the extra values are discarded.

To assign values from a list to the variables **a**, **b**, and **c**, the **for** clause could be used to bind the variable **numlist** to the **car** of the supplied form, and then another **for** clause could be used to bind the variables **a**, **b**, and **c** sequentially.

```
;; Collect values by using FOR constructs.
 (loop for numlist in '((1 2 4.0) (5 6 8.3) (8 9 10.4))
       for a integer = (first numlist)
       and b integer = (second numlist)
       and c float = (third numlist)
       collect (list c b a))
=> ((4.0 2 1) (8.3 6 5) (10.4 9 8))
```

Destructuring makes this process easier by allowing the variables to be bound in each loop iteration. Types can be declared by using a list of *type-spec* arguments. If all the types are the same, a shorthand destructuring syntax can be used, as the second example illustrates.

```
;; Destructuring simplifies the process.
 (loop for (a b c) (integer integer float) in
       '((1 2 4.0) (5 6 8.3) (8 9 10.4))
       collect (list c b a)))
=> ((4.0 2 1) (8.3 6 5) (10.4 9 8))


;; If all the types are the same, this way is even simpler.
 (loop for (a b c) float in
       '((1.0 2.0 4.0) (5.0 6.0 8.3) (8.0 9.0 10.4))
       collect (list c b a))
=> ((4.0 2.0 1.0) (8.3 6.0 5.0) (10.4 9.0 8.0))
```

If destructuring is used to declare or initialize a number of groups of variables into types, the loop keyword **and** can be used to simplify the process further.

```
;; Initialize and declare variables in parallel by using the AND construct.
(loop with (a b) float = '(1.0 2.0)
        and (c d) integer = '(3 4)
        and (e f)
        return (list a b c d e f))
=> (1.0 2.0 3 4 NIL NIL)
```

A type specifier for a destructuring pattern is a tree of type specifierswith the same shape as the tree of variables, with the following exceptions:

- When aligning the trees, an atom in the type specifier tree that matches a cons in the variable tree declares the same type for each variable.

- A cons in the type specifier tree that matches an atom in the variable tree is a nonatomic *type-specifer*.

If **nil** is used in a destructuring list, no variable is provided for its place.

```
(loop for (a nil b) = '(1 2 3)
      do (return (list a b)))
=> (1 3)
```

Note that nonstandard lists can specify destructuring.

```
(loop for (x . y) = '(1 . 2)
      do (return y))

(loop for ((a . b) (c . d)) ((float . float) (integer . integer)) in
      '(((1.2 . 2.4) (3 . 4)) ((3.4 . 4.6) (5 . 6)))
      collect (list a b c d))
=> ((1.2 2.4 3 4) (3.4 4.6 5 6))
```

An error of type **conditions:program-error** is signalled if the same variable is bound twice in any variable-binding clause of a single **future-common-lisp:loop** expression. Such variables include local variables, iteration control variables, and variables found by destructuring.

Understanding Compatibility Issues

Lisp Dialects Available in Genera

Genera provides four 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 (SCL) is the default dialect in Genera.

Zetalisp
: The dialect of Lisp provided in all Symbolics releases prior to Genera 7.0.

Common Lisp
: An implementation of Common Lisp as described in *Common Lisp: the Language (CLtL)*, by Guy Steele.

CLtL
: A strict, even harsh, implementation of Common Lisp as described in *Common Lisp: the Language (CLtL)*, by Guy Steele. This dialect is useful when developing programs with the intention of porting them to other implementations of Common Lisp. For information on how to use this dialect, see the section "Developing Portable Common Lisp Programs".

All of these dialects have some underlying features in common:

- Both the interpreter and the compiler use lexical scoping.
- Characters are represented as character objects.
- Row-major arrays are used.

Both Symbolics Common Lisp and Zetalisp use the interpreter, the 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 *Common Lisp: the Language* (*CLtL*) by Guy Steele,see the section "Compatibility with Common Lisp".

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 Common Lisp 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.

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 documen-

tation for information about these features. See the section "Functions in the CL Package with SCL Extensions". See the section "SCL-Specific Language Extensions".

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 -*-
```

Compatibility with Common Lisp

Some differences exist between the Symbolics implementation of Common Lisp in Genera and the language specification presented in Guy Steele's *Common Lisp: The Language* manual *(CLtL)*. This section contains tables listing Symbolics extensions, incompatible functions, and implementation decisions that might be of interest to you with regard to portability. When writing portable programs, use this list as a guide to help you.

Overview:

Guy Steele's book, *Common Lisp: The Language* (*CLtL*), describes what is required of a Common Lisp implementation. It also leaves room for (and in some cases encourages or even requires) implementation-dependent extensions.

In situations where portability of Common Lisp code is important, you should take care to avoid the use of any extensions, or to make sure that matching extensions are available in the implementations to which you intend to port your code.

To screen out most of the Symbolics extensions, you can simply use the **lisp** package, rather than the **scl** package. However, because some function names are shared between the **lisp** and **scl** packages, certain extended features are visible even when you use only the **lisp** package. In this section, we survey some common compatibility problems.

Extensions are called *compatible* if programs written to use only what is promised by *CLtL* are not adversely affected by the extension. For example, if function *f* were documented by *CLtL* to take only one argument, and our environment extended it to take a second optional argument, the extension would be called compatible as long as the one-argument use of *f* conformed to the *CLtL* standard.

Here is an overview of the tables included in this chapter:

- Functions in the Common Lisp package with Symbolics Common Lisp (SCL) extensions.
- SCL-specific language extensions.
- Functions in SCL that are incompatible with Common Lisp, as specified in *CLtL*.
- Incompatible language implementations.
- Compatible Differences and Clarifications with Common Lisp.
- Using package prefixes in portable programs.

Functions in the CL Package with SCL Extensions

Some functions in the **common-lisp** package accept additional optional or keyword arguments. We call these optional or keyword arguments *extensions*.

The following is a list of functions in the **common-lisp** package that have been extended in this way, and the names of the additional arguments that these functions accept. These extensions are compatible.

The function name is listed in the left column, in bold, and the Symbolics Common Lisp extension is in the right column, in italics.

Function	*Extension(s)*
adjoin	*:area*, *:localize*, *:replace*
adjust-array	*:displaced-conformally*
apropos	*do-inherited-symbols*, *do-packages-used-by*
apropos-list	*do-packages-used-by*
assoc-if	*:key*
assoc-if-not	*:key*
cerror	*optional-condition-name*
compile-file	*:package*, *:load*, *:set-default-pathname*
copy-alist	*area*
copy-list	*area*, *force-dotted*

copy-seq	*area*
delete-duplicates	*:replace* *:replace* is not meaningful if *:from-end t* is also used.
describe	*no-complaints*
disassemble	*from-pc*, *to-pc*
dribble	*editor-p*
eval	*env*
functionp	*allow-special-forms*
get-setf-method	*for-effect*
get-setf-method-multiple-value	*for-effect*
macroexpand	*dont-expand-special-forms*
macroexpand-1	*dont-expand-special-forms*
make-array	*:displaced-conformally*, *:area*, *:leader-list*, *:leader-length*, *:named-structure-symbol*
make-hash-table	*:area*, *:hash-function*, *:rehash-before-cold*, *:rehash-after-full-gc*, *:entry-size*, *:number-of-values*, *:store-hash-code*, *:mutating*, *:initial-contents*, *:optimizations*, *:locking*, *:ignore-gc*, *:growth-factor*, *:growth-threshold*
make-list	*:area*
make-package	*:prefix-name*, *:shadow*, *:export*, *:import*, *:shadowing-import*, *:import-from*, *:relative-names*, *:relative-names-for-me*, *:size*, *:external-only*, *:new-symbol-function*, *:hash-inherited-symbols*, *:invisible*, *:colon-mode*, *:prefix-intern-function*, *:include*
make-sequence	*:area*
make-symbol	*permanent-p*
make-string	*:element-type*, *:area*
pathname	*defaults*
push	*:area*, *:localize*
pushnew	*:area*, *:localize*, *:replace*
rassoc-if	*:key*
rassoc-if-not	*:key*
sleep	*:sleep-reason*

time *describe-consing*

write *:array-length*, *:string-length*, *:bit-vector-length*, *:abbreviate-quote*, *:readably*, *:string-length*, *:structure-contents*

SCL-Specific Language Extensions

Some functions, macros, and objects have extended semantics in Symbolics Common Lisp (SCL). For example, a function might be defined in SCL to have argument types not defined in *Common Lisp*: *the Language* (*CLtL*).

Following is a list of functions in the **lisp** package that treat their arguments in an extended way. These extensions are compatible.

Each bulleted item contains a description of the language extension, and a reference to the chapter or section in *CLtL* that discusses the topic.

- All atoms (non-lists) that are not symbols are self-evaluating, although *CLtL* only requires that bit-vectors, numbers, characters, and strings be so. See *CLtL*: Chapter 2, Data Types.

- **apply** is extended to allow you to call an array as a function. *CLtL* does not specify this case. See *CLtL*: Section 7.3, Function Invocation.

- **funcall** is extended to allow you to call an array as a function, with indices as arguments. *CLtL* does not specify this case. See *CLtL*: Section 7.3, Function Invocation.

- **if** is extended to allow you to supply more than three subforms; *CLtL* defines only the cases of two, or three, subforms. See *CLtL*: Section 7.6, Conditionals. **loop** allows many SCL extensions. *CLtL* does not define the meaning of atoms at the top-level of **loop**, so only the following is defined:

  ```
  (loop ...non-atomic-forms)
  ```

  You should avoid using **loop**'s keyword extensions in portable code, unless the target implementation is known to provide a compatible extension. For a list of these extensions, see the the *Flow of Control* chapter. See *CLtL*: Section 7.8.1, Indefinite Iteration.

- **shadow** is extended to accept string arguments, in addition to the symbol arguments specified in *CLtL*. See *CLtL*: Section 11.7, Package System Functions and Variables.

- All arrays are adjustable in SCL. *CLtL* promises only that arrays created with **:adjustable t** will be adjustable. *CLtL* does not specify what will happen if you omit **:adjustable**, or even what will happen if you specify **:adjustable nil**. See *CLtL*: Section 17.1, Array Creation.

- In Genera the value of (**char-code-limit**) is 65536. *CLtL* does not specify a value for this limit, but many users expect the limit to be considerably smaller (for example, 128 or 256), and are surprised by the storage used when they do things like `(make-array char-code-limit)`. If you are considering a portable lookup table for characters, you might want to consider using a hash table, rather than an array. See *CLtL*: Section 13.1, Character Attributes.

- In Genera, a displaced array need not be of the same type as the array to which it is displaced. *CLtL* says that it is an error if a displaced array is not the same type as the array to which it is displaced. See *CLtL*: Section 17.1, Array Creation.

- The following string functions are extended to accept character arguments, in addition to the argument types string and symbol, which are specified by *CLtL*:

string=	**string-equal**
string<	**string-lessp**
string>	**string-greaterp**
string<=	**string-not-greaterp**
string>=	**string-not-lessp**
user::string//=	**string-not-equal**
string-trim	**string-upcase**
string-right-trim	**string-downcase**
string-left-trim	**string-capitalize**

  See *CLtL*: Chapter 18, Strings.

- The value of ***print-pretty*** can take on values other than those defined by *CLtL*. See *CLtL*: Section 22.2.6, What the Print Function Produces.

- **open** is extended in a number of ways:

  Genera supports additional keywords from some file hosts. For a list of these extensions, see the section "SCL Functions Incompatible with Common Lisp".

  CLtL defines a fixed set of keywords for **open**: **:direction**, **:element-type**, **:if-exists**, and **:if-does-not-exist**. The Genera implementation accepts additional keywords. See *CLtL*: Section 23.2, Opening and Closing Files.

- **error** is extended so that its first argument may be other than just a string. The first argument may be a condition type, in which case the rest of the arguments are taken as init keywords for that condition, or the first argument may be a condition object, in which case there should be no additional arguments. See *CLtL*: Chapter 24, Errors.

- **documentation** is extended to allow documentation types other than those named in *CLtL*. See *CLtL*: Section 25.2, Documentation.

SCL Functions Incompatible with Common Lisp

This table lists functions in SCL that are incompatible with Common Lisp, as specified in *Common Lisp: the Language* (*CLtL*).

If you write programs that you plan to port to other systems, do not use these functions.

Each item in this list includes a reference to the section in *CLtL* that discusses the topic.

In the left column is the function name, in bold, and in the right column is an explanation of the incompatibility.

Function	*Reason for Incompatibility*
applyhook	The SCL variable **applyhook** is special, initially bound (its value cell is linked to that of ***applyhook***,) and is dangerous to assign without a clear understanding of what that assignment will affect. *CLtL* makes no claims about this name being used as a variable, so you might reasonably expect that this is a variable name available for normal purposes. Unfortunately, it is not. See *CLtL*: Section 20.1, Run-Time Evaluation of Forms.
char-equal	Does not ignore bits; *CLtL* specifies that this predicate should ignore bits. See *CLtL*: Section 13.2, Predicates on Characters.
char-not-equal	Does not ignore bits; *CLtL* specifies that this predicate should ignore bits. See *CLtL*: Section 13.2, Predicates on Characters.
char-lessp	Does not ignore bits; *CLtL* specifies that this predicate should ignore bits. See *CLtL*: Section 13.2, Predicates on Characters.
char-greaterp	Does not ignore bits; *CLtL* specifies that this predicate should ignore bits. See *CLtL*: Section 13.2, Predicates on Characters.
describe	Returns one value (its argument) instead of none. See *CLtL*: Section 25.3, Debugging Tools.
evalhook	The SCL variable **evalhook** is special, initially bound (its value cell is linked to that of ***evalhook***,) and is dangerous to assign without a clear understanding of what that assignment will affect. *CLtL* makes no claims about this name being used as a variable, so you might reasonably expect that this is a variable name available for normal purposes. Unfortunately, it is not. See *CLtL*: Section 20.1, Run-Time Evaluation of Forms.
functionp	Returns **t** if the argument is a symbol *and* if the argument, when the function **fbound** is used on it, returns **t**, (**fbound** returns true for that symbol), or else it returns **nil**. *CLtL* says **functionp** returns **t** for *any* symbol, whether it also returns true if **fbound** is used on it. See *CLtL*: Section 6.2.2, Specific Data Type Predicates.

gethash Returns three values instead of two. See *CLtL*: Section 16.1, Hash Table Functions.

make-echo-stream This function is not available in the Symbolics implementation of Common Lisp, because its contract is not well enough defined in our stream system. See *CLtL*: Section 21.1, Creating New Streams.

open The implementation of this function is different from the specification in *CLtL* in a number of ways:

- *CLtL* defines a fixed set of keywords for **open**: **:direction**, **:element-type**, **:if-exists**, and **:if-does-not-exist**. The Genera implementation accepts additional keywords. This compatible difference was mentioned in the section "SCL-Specific Language Extensions"

- *CLtL* says that the default **:element-type** for **open** is **string-char**. In the Genera implementation the default **:element-type** is **character**. This difference is incompatible.

- *CLtL* says that the only valid values of the keyword **:direction** are: **:input**, **:output**, **:io**, **:probe** and **:direct**. Genera accepts a number of other values for this argument, such as **:in** and **:out**, and device-specific values such as **:block**. This compatible difference was mentioned in the section "SCL-Specific Language Extensions"

- Genera does not support all of the **:element-type** options for **open** promised by *CLtL*. *CLtL* says that **:element-type** accepts the following types: **string-char**, (**unsigned-byte** *n*), **unsigned-byte**, (**signed-byte** *n*) **signed-byte**, **character**, **bit**, (**mod** *n*), and **:default**. Specifically:

Element Type	*Status*
string-char	Supported by Genera (but is not the default).
character	Supported by Genera (and is the default).
unsigned-byte	Not supported by Genera.
(**unsigned-byte** 8)	Supported by Genera.
(**unsigned-byte** 16)	Supported by Genera.
(**unsigned-byte** 32)	Supported by Genera for FEP files only.
(unsigned-byte *n*)	Not supported by Genera (except for indicated special cases).
signed-byte	Not supported by Genera.

(signed-byte *n***)**	Not supported by Genera.
bit	Not supported by Genera.
(mod *n***)**	Not supported by Genera (due to a bug).
:default	Supported by Genera.

See *CLtL*: Section 23.2, Opening and Closing Files.

read-line Returns up to four values. *CltL* specifies that **read-line** return only two values. See *CLtL*: Section 22.2.1, Input From Character Streams.

unintern SCL specifies that this function's second argument defaults to *symbol-package*; *CLtL* specifies that the second argument defaults to *package*. See *CLtL*: Section 11.7, Package System Functions and Variables.

Other Incompatible Differences

Some functions, macros, and objects in Symbolics Common Lisp (SCL) have implementation specification that are incompatible with Common Lisp, as specified in *Common Lisp: The Language* (*CLtL*).

The following is a list of implementation specifications made in SCL, which are incompatible with Common Lisp, as specified in *CLtL*.

Each item in this list includes a reference to the section in *CLtL* that discusses the topic.

- The argument list for **&rest** parameters has dynamic extent.

  Furthermore, the list of arguments should not be modified destructively with the **rplaca** or **rplacd** functions. If you want to save or return an **&rest** argument, use the **copy-list** function first. See the lambda list keyword **&rest**. See *CLtL*: Section 5.2.2, Lambda Expressions.

- The declarations **type**, **ftype**, and **optimize** are not implemented in SCL. In general, most Common Lisp declarations other than **special** are ignored. See *CLtL*: Section 9.1, Declaration Specifiers.

- In Symbolics Common Lisp, as a convenience, any variable named **ignore** or **ignored** is treated as if it had an implicit **ignore** declaration. As any use of an ignored variable is an error, using a variable named **ignore** or **ignored** generates compiler warnings and may generate incorrect code (because the compiler may assume that the variable will never be used). Thus, code that references variables named **ignore** or **ignored** may run differently in Symbolics Common Lisp than in other implementations of Common Lisp.

- The scoping rules for **special** declarations in SCL are compatible with Zetalisp, and incompatible with the scoping rules specified in *CLtL*. Some programs will be affected by this incompatibility, but most will not.

  The following three examples illustrate the differences between SCL and the language specification in *CLtL*. These examples assume there is no proclamation of **b** as a **special** variable.

  Example 1:

  ```
  (setq b 0)

  (defun foo ()
    (let ((b 1))
      (let ((b b))
        (declare (special b))
        b)))
  ```

  (foo) returns 1 in SCL, but returns 0 in Common Lisp. In Common Lisp, the declaration of **b** applies to both references to **b** in **(let ((b b))**, but in SCL it applies only to the first reference (the binding), not the second (the initial value form).

  Example 2:

  ```
  (defun foo ()
    (let ((b 1))
      (let ((a b))
        (declare (special b))
        (+ a b))))
  ```

  (foo) returns 1 in SCL, but returns 0 in Common Lisp. In **(+ a b)**, **a** is 1 and **b** is 0 in SCL, but both **a** and **b** are 0 in Common Lisp. In **(let ((a b))**, **b** refers to the local variable **b** in SCL, but refers to the **special** variable **b** in Common Lisp. In **(+ a b)**, **b** refers to the special variable in both dialects.

  Example 3:

  ```
  (defun foo ()
    (let ((b 1))
      (declare (special b))
      (let ((b 2))
        (locally (declare (special b))
                 b))))
  ```

  (foo) returns 2 in SCL, but 1 in Common Lisp. In SCL **special** declarations are pervasive, so the first declaration of **b** affects both **let** forms. In Common Lisp, **special** declarations are not pervasive for bindings, so only the **(let ((b 1))** is affected. This example issues a compiler warning about the incompatibility. This example also illustrates a bug in the SCL interpreter, which is compatible with Common Lisp, rather than with the SCL compiler, in this case.

See *CLtL*: Section 9.1 and 9.2, Declaration Syntax and Declaration Specifiers.

- SCL supports packages that are unavailable in other implementations. *CLtL* supports the **user**, **keyword**, **system**, and **lisp** packages. See *CLtL*: Chapter 11, Packages

- Package-name lookup is not case-sensitive. See *CLtL*: Section 11.3, Translating Strings to Symbols.

- The constructor does not evaluate **defstruct** slot initializations in the appropriate lexical environment. See *CLtL*: Section 19.6, By-position Constructor Functions.

- A top-level form that returns no values does not set the variable *. The variable * remains unchanged. See *CLtL*: Section 20.2, The Top-Level Loop.

- Setting the value of ***read-base*** to 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 **sys:*read-extended-ibase-signed-number*** and **sys:*read-extended-ibase-unsigned-number*** to **t** to cause the tokens to always be interpreted as numbers. See *CLtL*: Section 21.1.2, Parsing of Numbers and Symbols.

- 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. See *CLtL*: Section 21.1.5, The Readtable.

- SCL 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. See *CLtL*: Section 21.1.2, Parsing of Numbers and Symbols.

- Symbols in the ***features*** list must be keywords in order 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. See *CLtL*: Section 22.1.4, Standard Dispatching Macro Character Syntax.

- In SCL, the concept of alphabetic case is not meaningful for a character with a non-zero bits field. Instead, SCL has the concept of a Shift bit, which is only meaningful for alphabetic characters with non-zero control, meta, super or hyper bits. The Shift bit indicates whether the Shift key was pressed when the character was typed on the keyboard. Note that the Shift bit is not affected by the Caps Lock key, nor by any software caps lock, for example Zmacs' Electric Shift Lock mode. This is why the Shift bit is not the same as alphabetic case.

In SCL, however, the printed representations **#\control-meta-A** and **#\control-meta-\a** both read as the same character. The character whose printed representation is **#\control-meta-shift-A** is produced by **(make-char #\a (+ char-control-bit char-meta-bit))**.

This difference is not considered a serious portability problem, since the availability of character modifier bits on keyboards is implementation-dependent. See *CLtL*: Section 22.1.6, What the Print Function Produces.

- **#\newline** prints as **#\Return**. See *CLtL*: Section 22.1.6, What the Print Function Produces.

- 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 *CLtL*. See *CLtL*: Section 22.1.6, What the Print Function Produces.

- 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. See *CLtL*: Section 23.1.2, Pathname Functions.

- **load** uses ***load-pathname-defaults*** as the default for *filename*, rather than ***default-pathname-defaults***. ***load-pathname-defaults*** is a Zetalisp defaults-alist, whereas ***default-pathname-defaults*** is a Common Lisp default pathname.

  load ignores the **:print** argument.

  See *CLtL*: Section 23.5, Loading Files.

- The following functions can signal an error such as "no directory":

  file-author
  file-length
  file-namestring
  file-position
  file-write-date
  probe-file

  See *CLtL*: Section 23.3, Renaming, Deleting, and Other File Operations

- The following functions may not work with all kinds of streams:

make-broadcast-stream
make-string-input-stream
make-string-output-stream
make-synonym-stream
make-two-way-stream

See *CLtL*: Chapter 21, Streams.

- The function **dribble** makes a new read-eval-print loop. It does not work by side-effect, as do some other Common Lisp implementations. See *CLtL*: Section 25.3, Debugging Tools.

Compatible Differences and Clarifications

The following is a list of implementation decisions made in SCL when *Common Lisp: The Language* (*CLtL*) left unspecified the implementation of certain functions.

If you plan to write programs to port to other systems, you should be aware of these implementation differences. If you are unaware of these differences, you could write a program that relies on some functionality that will not port to another system.

Most of these implementation decisions are compatible; incompatible cases are indicated.

Each bulleted item includes a description of the differences between our implementation and that of *CLtL*, and contains references the chapter or section in *CLtL* that discusses the topic.

- The type **string** is implemented as a subtype of the type **common**. The type **string-char** is not a subtype of the type **common**.

  standard-char is a subtype of **string-char**, but **string-char** is not a subtype of **standard-char**. See *CLtL*: Section 2.2.5, String Characters.

- Some uses of the **coerce** function signal errors. This is not an incompatibility with *CLtL*; the following examples are provided only for clarification:

  (coerce '(1 2 3) '(vector t 3))

  Signals an error because the length is specified.

  (coerce #\c-A 'string-char)

  Signals an error because **coerce** cannot coerce to a **string-char**.

  (coerce 22/7 '(float 0 10))

  Signals an error because **coerce** cannot coerce to a subrange of floats. See *CLtL*: Section 4.8, Type Conversion Function.

- The compiler ignores the *value-type* argument in the **the** special form. See *CLtL*: Section 9.1, Declaration Syntax.

- The functions **rename-package**, **intern**, and **find-symbol** allow package names in places where *CLtL* requires packages. See *CLtL*: Section 11.7, Package System Functions and Variables.

- The **substitute**, **substitute-if**, and **substitute-if-not** functions are not optimized for detecting the case in which they can return just their argument. See *CLtL*: Section 14.3, Modifying Sequences.

- All sequence and list functions that take a two-argument predicate (such as **:test** and **:test-not**) 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. See *CLtL*: Chapter 14, Sequences, and Chapter 15, Lists.

- **#p** is used for printing pathnames and is followed by a string in double quotes.

  Common Lisp does not specify a specific syntax for printing objects of type **pathname**. However, every implementation must arrange to print a pathname in such a way that, within the same implementation of Common Lisp, the function **read** can construct from the printed representation an equivalent instance of the pathname object. See *CLtL*: Section 22.1.3, Macro Characters.

- The **read-char** function echoes the character read from the input stream if it is the terminal. See *CLtL*: Section 22.2.3, Formatted Output to Character Streams.

- 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. See *CLtL*: Section 22.2.3, Formatted Output to Character Streams.

- The ~T directive does not know the column position when the output is directed to a file. See *CLtL*: Section 22.2.3, Formatted Output to Character Streams.

- 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. See *CLtL*: Section 23.5, Accessing Directories.

- 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. See *CLtL*: Section 25.3, Debugging Tools.

Using Package Prefixes in Portable Programs

Using the cl: Package Prefix

Using the package prefix **cl:** in your portable programs can cause compatibility problems. If you make a package that shadows any or all Lisp symbols, and you then try to print some symbol from the Lisp package, it prints as **cl:***symbol-name*. *CLtL* says that it should print as **lisp**:*symbol-name*.

If you are writing a portable program, and a package prefix is needed, you should always write **lisp**:*symbol-name*, rather than **cl:***symbol-name*, since the compiler recognizes the package name **lisp.**

Using a Genera-Specific Package

SCL supports some packages that are unavailable in other systems. *CLtL* commits to supporting the following packages: **user**, **keyword**, **system**, and **lisp**. If you write programs that you plan to port to other systems, you should not use functions in a SCL-specific package.

See *CLtL*: Chapter 11, Packages.