Structure Macros

This section contains reference information on the use of **defstruct**, **future-common-lisp:defstruct**, and **zl:defstruct**. For an overview of structure macros: See the section "Overview of Structure Macros".

Basic Use of defstruct

Genera provides three **defstruct** symbols:

defstruct — The Symbolics Common Lisp symbol, which offers many extensions to the **defstruct** as specified by Common Lisp.

future-common-lisp:defstruct — This macro adheres to the draft ANSI Common Lisp specification. You can define CLOS methods that specialize on instances of structure classes defined by **future-common-lisp:defstruct**.

zl:defstruct — From Zetalisp, and provided for compatibility reasons.

defstruct *options* &body *items* *Macro*

Defines a record-structure data type. A call to **defstruct** looks like:

```
(defstruct (name option-1 option-2 ...)
           slot-description-1
           slot-description-2
           ...)
```

name must be a symbol; it is the name of the structure. It is given a **si:defstruct-description** property that describes the attributes and elements of the structure; this is intended to be used by programs that examine other Lisp programs and that want to display the contents of structures in a helpful way. *name* is used for other things; for more information, see the section "Named Structures".

Because evaluation of a **defstruct** form causes many functions and macros to be defined, you must take care not to define the same name with two different **defstruct** forms. A name can only have one function definition at a time. If a name is redefined, the later definition is the one that takes effect, destroying the earlier definition. (This is the same as the requirement that each **defun** that is intended to define a distinct function must have a distinct name.)

Each *option* can be either a symbol, which should be one of the recognized option names, or a list containing an option name followed by the arguments to the option. Some options have arguments that default; others require that arguments be given explicitly. For more information about options, see the section "Options for **defstruct**".

Each *slot-description* can be in any of three forms:

```
1:   slot-name
2:   (slot-name default-init)
3:   ((slot-name-1 byte-spec-1 default-init-1)
      (slot-name-2 byte-spec-2 default-init-2)
                ...)
```

Each *slot-description* allocates one element of the physical structure, even though several slots may be in one form, as in form 3 above.

Each *slot-name* must always be a symbol; an accessor function is defined for each slot.

In the example above, form 1 simply defines a slot with the given name *slot-name*. An accessor function is defined with the name *slot-name*. The **:conc-name** option allows you to specify a prefix and have it concatenated onto the front of all the slot names to make the names of the accessor functions. Form 2 is similar, but allows a default initialization for the slot. Form 3 lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of **defstruct**.

For a table of related items: See the section "Functions Related to **defstruct** Structures".

future-common-lisp:defstruct *name-and-options* &body *slot-descriptions* *Macro*

Defines a record-structure data type, and a corresponding class of the same name. You can define methods that specialize on structure classes.

The syntax and semantics of **future-common-lisp:defstruct** adhere to the draft ANSI Common Lisp specification.

zl:defstruct *Macro*

Defines a record-structure data type. Use the Common lisp macro **defstruct**. **defstruct** accepts all standard Common Lisp options, and accepts several additional options. **zl:defstruct** is supported only for compatibility with pre-Genera 7.0 releases. See the section "Differences Between **defstruct** and **zl:defstruct**".

The basic syntax of **zl:defstruct** is the same as **defstruct**: See the macro **defstruct**.

For information on the options that can be given to **zl:defstruct** as well as **defstruct**: See the section "Options for **defstruct**".

The **:export** option is accepted by **zl:defstruct** but not by **defstruct**. Stylistically, it is preferable to export any external interfaces in the package declarations instead of scattering **:export** options throughout a program's source files.

:export

Exports the specified symbols from the package in which the

structure is defined. This option accepts as arguments slot names and the following options: **:alterant**, **:accessors**, **:constructor**, **:copier**, **:predicate**, **:size-macro**, and **:size-symbol**.

The following example shows the use of **:export**.

```
(zl:defstruct (2d-moving-object
                (:type :array)
                :conc-name
                ;; export all accessors and the
                ;; make-2d-moving-object constructor
                (:export :accessors :constructor))
  mass
  x-pos
  y-pos
  x-velocity
  y-velocity)
```

See the section "Importing and Exporting Symbols".

Options for defstruct

This section describes the options that can be given to **defstruct** and **zl:defstruct**. The description of each option states any differences in behavior of the option, when given to **defstruct** and **zl:defstruct**.

Note: The **:export** option can be given to **zl:defstruct** but not **defstruct**. It is described elsewhere: See the macro **zl:defstruct**.

Here is an example that shows the typical syntax of a call to **defstruct** that gives several options.

```
(cl:defstruct (foo (:type vector)
                   :conc-name
                   (:size-symbol foo))
  a
  b)
```

:type

Specifies the kind of Lisp object to be used to implement the structure. The option requires one argument, which must be one of the symbols enumerated below, or a user-defined type. If the option itself is not provided, the type defaults to **:array**. You can define your own types by using **defstruct-define-type**.

The **:type** option can be given to both **defstruct** and **zl:defstruct**, but they accept different arguments.

These arguments are accepted by the **defstruct :type** option, but not by the **zl:defstruct :type** option:

vector

Use a vector, storing components as vector elements.

If the structure is **:named**, element 0 of the vector holds the named structure symbol and is therefore not used to hold a component of the structure.

You can use the **:make-array** option with **(:type vector)** to specify the area in which the structures should be made. For example:

```
(defstruct
  (foo (:type vector)
       (:make-array (:area *foo-area*)))
  x y z)
```

(vector *element-type***)**

Use a vector, storing components as vector elements. Each component must be of a type that can be stored in a vector of *element-type*. The structure may be **:named** only if the type **symbol** is a subtype of the specified *element-type*.

If the structure is **:named**, element 0 of the vector holds the named structure symbol and is therefore not used to hold a component of the structure.

These arguments are accepted by the **defstruct :type** option and the **zl:defstruct :type** option:

list

Use a list, storing components as list elements.

If the structure is **:named**, the car of the list holds the named structure symbol and is therefore not used to hold a component of the structure.

You can use the **:make-list** option with **(:type list)** to specify further options about the list that implements the structure.

:list

Same as the **list** option for **defstruct**.

:named-list

Like **:list**, but the first element of the list holds the symbol that is the name of the structure, and so is not used as a component.

:array

Use an array, storing components in the body of the array.

:named-array

Like **:array**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures". Element 0 of the array holds the named structure symbol and is therefore not used to hold a component of the structure.

:array-leader

Use an array, storing components in the leader of the array. See the section "Options for **defstruct**".

:named-array-leader

Like **:array-leader**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures". Element 1 of the leader holds the named structure symbol and so is not used to hold a component of the structure.

:tree

Implements structure out of a binary tree of conses, with the leaves serving as the slots.

:grouped-array

See the section "Grouped Arrays". This option is described there.

:alterant

Allows you to customize the name of the alterant function. If **(:alterant** *name***)** is supplied, the name of the alterant function is *name*. *name* should be a symbol; its print name is the name of the alterant function.

If **:alterant** is specified without an argument, the name of the alterant is **alter-***structure*. This is also the default behavior of **zl:defstruct**, when the **:alterant** option is not given.

If **(:alterant nil)** is specified, no alterant is defined. This is also the default behavior of **zl:defstruct**, when the **:alterant** option is not given.

The following example defines the alterant to be **change-door-slot**.

```
(cl:defstruct (door (:alterant change-door-slot))
  knob-color width)

(setq d (make-door :knob-color 'red :width 5.0))

(change-door-slot d
   knob-color 'blue
   width 5.5)
```

For more information on the use of the alterant macro: See the section "Alterant Macros for **defstruct** Structures".

defstruct and zl:defstruct Difference

defstruct and **zl:defstruct** have different default behavior when **:alterant** is not supplied:

defstruct	Does not define an alterant.
zl:defstruct	Defines an alterant named **alter-***structure*.

:but-first

The argument to **:but-first** is an accessor from some other structure, and it is expected that this structure will never be found outside that slot of that other structure. Actually, you can use any one-argument function, or a macro that acts like a one-argument function. It is an error for **:but-first** to be used without an argument.

This example should clarify the use of **:but-first**.

```
(cl:defstruct (head (:type list)
                    (:default-pointer person)
                    (:but-first person-head))
  nose
  mouth
  eyes)
```

The **nose** accessor expands like this:

```
(nose x)        => (car (person-head x))
(nose)          => (car (person-head person))
```

:callable-accessors

This option controls whether accessors are really functions, and therefore "callable", or whether they are macros.

The accessors are functions if this option is not provided, provided with no argument, or provided with an argument of **t**. Specifically, they are substs, so that they have all the efficiency of macros in compiled programs, while still being function objects that can be manipulated (passed to **mapcar**, and so on).

If this option is provided with an argument of **nil**, then the accessors will be macros, not substs.

Note that if you use the **:default-pointer** option, the accessors cannot be made callable.

:conc-name

Allows you to customize the names of the accessor functions. If **(:conc-name** *prefix***)** is supplied, the name of each accessor function is *prefix-slot*. *prefix* should be a symbol; its print name is concatenated onto the front of all the slot names to make the names of the accessor functions.

If **:conc-name** is specified without an argument, the name of each accessor is *structure-slot*; that is, the name of the structure followed by a hyphen, followed by the slot name. This is also the default behavior of **defstruct**, when the **:conc-name** option is not given.

:conc-name changes the name of the accessor functions, but has no effect on slot names that are given to the constructor and alterant macros. Thus when you use **:conc-name**, the slot names and accessor names are different.

In the following example, the **:conc-name** option specifies the prefix "**get-door-**", which causes the accessor functions to be named **get-door-knob-color** and **get-door-width**.

```
(cl:defstruct (door (:conc-name get-door-))
  knob-color
  width)

(setq d (make-door :knob-color 'red :width 5.0))

(get-door-knob-color d) => red
```

If **(:conc-name nil)** is specified, the name of each accessor is *slot*, the name of the slot. This is also the default behavior of **zl:defstruct**, when the **:conc-name** option is not given. When the name of the accessor is just *slot*, you should name the slots according to a suitable convention. You should always prefix the names of all accessor functions with some text unique to the structure.

defstruct and zl:defstruct Difference

defstruct and **zl:defstruct** have different default behavior when **:conc-name** is not supplied:

defstruct	Names each accessor *structure-slot*.
zl:defstruct	Names each accessor *slot*, the name of the slot.

:constructor

Takes one argument, which specifies the name of the constructor. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string **"make-"** to the name of the structure. If the argument is provided and is **nil**, no constructor is defined. A more general form of this option is also available: See the section "By-position Constructors for **defstruct** Structures".

For more information about the use of the constructor: See the section "Constructors for **defstruct** Structures".

defstruct and zl:defstruct Difference

defstruct	Defines a constructor function.
zl:defstruct	Defines a constructor macro.

:constructor-make-array-keywords

If the structure being defined is implemented as an array, this option can be used to take certain **zl:make-array** keywords as arguments, and determine them on an instance by instance basis. This is in contrast to the keyword option **:make-array**, which supplies **zl:make-array** keywords that apply to all instances of the structure.

For example, you can use this option to define a structure that is implemented as an array leader, and specify the length of the array after the leader elements.

The arguments to the **:constructor-make-array-keywords** option are **zl:make-array** keywords. The **:constructor-make-array-keywords** option lets you control the initialization of arrays created by **defstruct** as instances of structures. **zl:make-array** initializes the array before the constructor code does. Therefore, any initial value supplied via the new **:initial-value** keyword for **zl:make-array** is overwritten in any slots where you gave **defstruct** an explicit initialization.

Here is an example of a **defstruct** using **:constructor-make-array-keywords** to create an array whose **defstruct** slots are in the array-leader. Using this option, you can indicate how long the part of the array after the leader should be each time you make a new array-leader.

```
;; the name of the defstruct is make-logic-variable-environment
(make-logic-variable-environment :length 10)

To define this defstruct:
```

```
(defstruct (logic-variable-environment
             (:type :array-leader)
             :named
             (:constructor-make-array-keywords length))
  ;; place to stash wayward variables and some bits for later use.
  home-for-wayward-variables
   (bits (make-logic-variable-environment-bits)))
```

:copier

The **:copier** option allows you to customize the name of the copier function. If **(:copier** *name***)** is supplied, the name of the copier function is *name*. *name* should be a symbol; its print name is the name of the copier function.

The automatically defined copier function simply makes a new structure and transfers all components verbatim from the argument into the newly created structure. No attempt is made to make copies of the components. Corresponding components of the old and new structures are therefore **eql**.

If **:copier** is specified without an argument, the name of the copier function is **copy-***structure*. This is also the default behavior of **defstruct** when the **:copier** option is not given.

If **(:copier nil)** is specified, no copier is defined. This is also the default behavior of **zl:defstruct** when the **:copier** option is not given.

For example:

```
(cl:defstruct (foo (:type list) :copier)
  foo-a
  foo-b)
```

This example would generate a function named **copy-foo**, with a definition approximately like this:

```
(defun copy-foo (x)
  (list (car x) (cadr x)))
```

defstruct and zl:defstruct Difference

defstruct and **zl:defstruct** have different default behavior when **:copier** is not supplied:

defstruct	Defines a copier name **copy-***structure*.
zl:defstruct	Does not define a copier.

:default-pointer

Normally, the accessors defined by **defstruct** expect to be given exactly one argument. However, if the **:default-pointer** argument is used, the argument to each accessor is optional.

You can continue to use the accessor function in the usual way. You can also invoke an accessor without its argument; it behaves as if you had invoked it on the result of evaluating the form that is the argument to the **:default-pointer** argument. For example:

```
(cl:defstruct (room (:default-pointer *room-13*)
                    :conc-name)
  name
  contents)

(setq play-room
      (make-room :name 'den :contents 'tv))
(setq *room-13*
      (make-room :name 'kitchen :contents 'fridge))

(room-name play-room) => DEN
(room-name) => KITCHEN
```

If the argument to the **:default-pointer** argument is not given, it defaults to the name of the structure.

:eval-when

Normally, the functions and macros defined by **defstruct** are defined at eval time, compile time, and load time. This option allows you to control this behavior. The argument to the **:eval-when** option is just like the list that is the first subform of an **eval-when** special form. For example, **(:eval-when (eval compile))** causes the functions and macros to be defined only when the code is running interpreted or inside the compiler. Note that the default for **defstruct** is **(load eval)** .

:include

Allows you to build a new structure definition as an extension of an old structure definition. Suppose you have a structure called **person** that looks like this:

```
(defstruct (person)
   name
   age
   sex)
```

Now suppose you want to make a new structure to represent an astronaut. Since astronauts are people too, you would like them to also have the attributes of name, age, and sex, and you would like Lisp functions that operate on **person** structures to operate just as well on **astronaut** structures. You can do this by defining **astronaut** with the **:include** option, as follows:

```
(defstruct (astronaut (:include person))
   helmet-size
   (favorite-beverage 'tang))
```

The **:include** option inserts the slots of the included structure at the front of the list of slots for this structure. That is, an **astronaut** will have five slots; first, the three defined in **person**, then the two defined in **astronaut** itself. The accessor functions defined by the **person** structure can be applied to instances of the **astronaut** structure. The following illustrates how you can use **astronaut** structures:

```
(setq x (make-astronaut :name 'buzz
                        :age 45
                        :sex t
                        :helmet-size 17.5))

(person-name x) => buzz
(astronaut-favorite-beverage x) => tang
```

Note that the **:conc-name** option was *not* inherited from the included structure; it applies only to the accessor functions of **person** and not to those of **astronaut**. Similarly, the **:default-pointer** and **:but-first** options, as well as the **:conc-name** option, apply only to the accessor functions for the structure in which they are enclosed; they are not inherited if you include a structure that uses them.

The argument to the **:include** option is required, and must be the name of some previously defined structure of the same type as this structure. **:include** does not work with structures of type **:tree** or of type **:grouped-array**.

The following is an advanced feature. Sometimes, when one structure includes another, the default values for the slots that came from the included structure are not what you want. The new structure can specify different default values for the included slots than the included structure specifies, by giving the **:include** option as:

```
(:include name new-init-1 ... new-init-n)
```

Each *new-init* is either the name of an included slot or a list of the form (*name-of-included-slot init-form*). If it is just a slot name, the slot has no initial value in the new structure. Otherwise, its initial value form is replaced by the *init-form*. The old (included) structure is unmodified.

For example, to define **astronaut** so that the default age for an astronaut is **45**, the following can be used:

```
(defstruct (astronaut (:include person (age 45)))
   helmet-size
   (favorite-beverage 'tang))
```

:initial-offset

Allows you to tell **defstruct** to skip over a certain number of slots before it starts allocating the slots described in the body. This option requires an argument (which must be a fixnum) that is the number of slots you want **defstruct** to skip. To use this option, you must understand how **defstruct** is implementing your structure; otherwise, you will be unable to make use of the slots that **defstruct** has left unused.

:inline

Causes functions to be compiled inline. Values can be **:accessors**, **:constructor**, **:copier**, **:predicate**, or the name of a slot. Defaults to compiling accessors, constructors, and predicates inline. Note that the default is for most functions to be compiled inline. For example:

```
(:inline :constructor x-pos y-pos)
```

This example causes the constructor functions, **x-pos** and **y-pos**, to be compiled inline. For information on inline functions: See the section "Inline Functions".

:make-list

You can use the **:make-list** option with **(:type list)** or **(:type :list)** to specify further options about the list that implements the structure. For example, you can specify the area in which the structures should be made.

```
(defstruct
  (foo (:type list)
       (:make-list (:area *foo-area*)))
  x y z)
```

:make-array

If the structure being defined is implemented as an array, this option can be used to control those aspects of the array that are not otherwise constrained by **defstruct**. For example, you might want to control the area in which the array is allocated. Also, if you are creating a structure of type **:array-leader**, you almost certainly want to specify the dimensions of the array to be created, and you might want to specify the type of the array. Of course, this option is only meaningful if the structure is, in fact, being implemented by an array.

The argument to the **:make-array** option should be a list of alternating keyword symbols to the **zl:make-array** function (note that this is the Zetalisp version), and forms whose values are the arguments to those keywords. For example, **(:make-array**

(:area *foo-area*)) requests that structures of this type be consed in ***foo-area***. Note that the keyword symbol is not evaluated.

When necessary, **defstruct** overrides any of the **:make-array** options. For example, if your structure is of type **:array**, then **defstruct** supplies the size of that array, regardless of what you say in the **:make-array** option.

Constructor macros for structures implemented as arrays all allow the keyword **:make-array**. Attributes supplied therein override any **:make-array** option attributes supplied in the original **defstruct** form. If some attribute appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, the constructor chooses appropriate defaults.

The **:make-array** option lets you control the initialization of arrays created by **defstruct** as instances of structures. **zl:make-array** initializes the array before the constructor code does. Therefore, any initial value supplied via the new **:initial-value** keyword for **zl:make-array** is overwritten in any slots where you gave **defstruct** an explicit initialization.

If a structure is of type **:array-leader**, you probably want to specify the dimensions of the array. The dimensions of an array are given to **:make-array** as a position argument rather than a keyword argument, so there is no way to specify them in the above syntax. To solve this problem, you can use the keyword **:dimensions** or the keyword **:length** (they mean the same thing) with any value that **zl:make-array** accepts as a first argument.

:named

Allows you to use one of the "named" types. If you specify a type of **:array**, **:array-leader**, or **:list**, and give the **:named** option, then the **:named-array**, **:named-array-leader**, or **:named-list** type is used instead. Asking for type **:array** and giving the **:named** option as well is the same as asking for the type **:named-array**; the only difference is stylistic.

:predicate

Allows you to customize the name of the predicate function. The predicate function recognizes objects of this structure. If **(:predicate** *name***)** is supplied, the name of the predicate function is *name*. *name* should be a symbol; its print name is the name of the predicate function. The **:predicate** option works only for named types.

If **:predicate** is specified without an argument, the name of the predicate is *structure*-**p**. This is also the default behavior of **defstruct**, when the **:predicate** option is not given.

If **(:predicate nil)** is specified, no predicate is defined. This is also the default behavior of **zl:defstruct**, when the **:predicate** option is not given.

The following example defines a single-argument predicate function, **foo-p**, that returns **t** only for objects of structure **foo**.

```
(cl:defstruct (foo :named :predicate)
  foo-a
  foo-b)
```

The following example defines a predicate function called **is-it-a-foo?**.

```
(cl:defstruct (foo :named (:predicate is-it-a-foo?))
  foo-a
  foo-b)
```

defstruct and zl:defstruct Difference

The difference is in the default behavior, when **:predicate** is not supplied.

defstruct	If **:type** option is not given, or if both **:type** and **:named** are given, default is same as **:predicate** without an argument. If **:type** option is given and **:named** is not given, default is same as **(:predicate nil)**.
zl:defstruct	Default is same as **(:predicate nil)** regardless of whether the **:type** option is given.

:print

Has the same effect as the Common Lisp **:print-function** option. Gives you implementation-independent control over the printed representation of a structure. Using this option defeats the **sys:printing-random-object** mechanism. See the macro **sys:printing-random-object**.

The **:print** option takes a format string and its arguments. The arguments are evaluated in an environment in which the name symbol for the structure is bound to the structure instance being printed.

The **:print** option makes it unnecessary to use a **named-structure-invoke** handler to define **:print** handlers.

:print-function

Allows you to specify a function to be used to print this type of structure. The printer uses the print function for structures of unspecified type and when the type is specified as a named vector. The printer never uses a print function for a structure implemented as a named list, but the **describe-defstruct** function does.

The print function should accept three arguments: the structure to be printed, the stream, and an integer indicating the current depth. The function must be acceptable to the **function** special form.

The function must respect the following print control variables: ***print-escape***, ***print-pretty***, and ***print-structure-contents***.

You can use the function **sys:print-cl-structure** or the macro **sys:print-cl-structure** in a printer function. See the function **sys:print-cl-structure**. See the macro **sys:cl-structure-printer**.

```
(defun file-branch-print-function (b stream depth)
  (if *print-escape*
    (if *print-structure-contents*
       (sys:cl-structure-printer file-branch b stream depth)
       (sys:printing-random-object (b stream :typep)
         (format stream "~A" (file-branch-name b))))
    (format stream "~A" (file-branch-name b))))
```

Common Lisp specifies that **:print-function** may be used only if **:type** is not used; however, Genera does not enforce this restriction.

Note: The **:print-function** option is accepted by **defstruct** but not by **zl:defstruct**.

:property

For each structure defined by **defstruct**, a property list is maintained to record of arbitrary properties about that structure. (That is, there is one property list per structure definition, not one for each instantiation of the structure.)

The **:property** option can be used to give a **defstruct** an arbitrary property. **(:property** *property-name value***)** gives the **defstruct** a *property-name* property of *value*. Neither argument is evaluated. To access the property list, you should look inside the **si:defstruct-description** structure. See the section "**defstruct** Internal Structures".

:size-symbol

Allows you to specify a global variable whose value is the "size" of the structure; this variable is declared with **zl:defconst**. The exact meaning of the size varies, but in general, this number is the one you would need to know if you were going to allocate one of these structures yourself. The symbol has this value both at compile time and at run time. If this option is present without an argument, then the name of the structure is concatenated with **"-size"** to produce the symbol.

:size-macro

Similar to the **:size-symbol** option. A macro of no arguments is defined that expands into the size of the structure. The name of this macro defaults as with **:size-symbol**.

:times

Used for structures of type **:grouped-array**, to control the number of repetitions of the structure that are allocated by the constructor macro. The constructor macro also allows **:times** to be used as a keyword that overrides the value given in the original **defstruct** form. If **:times** appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor allocates only one instance of the structure.

type

In addition to the documented options to **defstruct** and **zl:defstruct**, any currently defined type (any valid argument to the **:type** option) can be used as an option. This is mostly for compatibility with older versions of **zl:defstruct**. It allows you to say just *type* instead of **(:type** *type***)**. It is an error to give an argument to one of these options.

other

Finally, if an option is not found among the other options, **defstruct** or **zl:defstruct** checks the property list of the name of the option to see if it has a non-**nil :defstruct-option** property. If it does have such a property, then if the option was of the form **(***option-name value***)**, it is treated just like **(:property** *option-name value***)**. That is, the structure is given an *option-name* property of *value*. It is an error to use such an option without a value.

This provides a primitive way for you to define your own options to **defstruct** or **zl:defstruct**, particularly in connection with user-defined types. See the section "Extensions to **defstruct**". Several options to **defstruct** and **zl:defstruct** are implemented using this mechanism.

defstruct Structures and type-of

Under certain circumstances, **defstruct** and **zl:defstruct** define the name of the structure as a type name in both the Common Lisp and Zetalisp type systems. In these circumstances it is illegal for the name of the structure to be the same as the name of an existing type (including a flavor or a built-in type).

The name of the structure is defined as a type name when the structure is defined in one of these ways:

- With **defstruct**, when the **:type** option is not given.

- With **defstruct**, when the **(:type :vector)** and **:named** options are given.
- With **defstruct**, when the **(:type (:vector** *element*)**)** and **:named** options are given.
- With **zl:defstruct**, when the **(:type :named-array)** option is given.
- With **zl:defstruct**, when the **(:type :array)** and **:named** options are given.
- With **zl:defstruct**, when the **(:type :named-array-leader)** option is given.
- With **zl:defstruct**, when the **(:type :array-leader)** and **:named** options are given.

When a structure is defined as a type name, **(type-of** *object*) returns the symbol that is the name of the object's structure.

(typep *object 'structure-name*) and **(zl:typep** *object 'structure-name*) return **t** if the flavor of *object* is named *structure-name*, **nil** otherwise.

Using the Constructor and Alterant Macros for defstruct Structures

The documentation in this section regarding **defstruct** also applies to **zl:defstruct**.

This section describes how to create instances of structures and alter the values of their slots. After you have defined a new structure with **defstruct**, you can create instances of this structure using the constructor, and you can alter the values of its slots using the alterant macro.

By default, **defstruct** defines a constructor function, forming its name by concatenating **"make-"** onto the name of the structure. If you use the **:alterant** option with no argument, an alterant macro is defined, its name formed by concatenating **"alter-"** onto the name of the structure.

You can specify the names of the constructor or alterant macros by passing the name you want to use as the argument to the **:constructor** or **:alterant** options. You can also specify that you do not want the macro created at all by passing **nil** as the argument.

Constructors for defstruct Structures

Note that **defstruct** implements the constructor as a function, but **zl:defstruct** implements it as a macro.

A call to a constructor has the form:

```
(name-of-constructor
        symbol-1 form-1
        symbol-2 form-2
        ...)
```

Each *symbol* indicates a slot of the structure (this is not necessarily the same as the name of the accessor). *symbol* can also be one of the specially recognized keywords described further on. If *symbol* indicates a *slot*, that element of the created structure is initialized to the value of the corresponding *form*. All the *forms* are evaluated.

When using the constructor for a **defstruct**-defined structure, the *symbol* that indicates a slot must be the name of that slot in the keyword package.

```
(cl:defstruct door1
   knob-color
   width)

(make-door1 :knob-color 'red    ;slot name in keyword package
            :width 5.5)
```

When using the constructor for a **zl:defstruct**-defined structure, the *symbol* that indicates a slot should just be the name of the slot.

```
(zl:defstruct door2
   knob-color
   width)

(make-door2 knob-color 'red    ;slot name
            width 5.5)
```

If no *symbol* is present for a given slot, the slot is initialized to the result of evaluating the default initialization form specified in the call to **defstruct**. In other words, the initialization form specified to the constructor overrides the initialization form specified to **defstruct**. If the **defstruct** itself also did not specify any initialization, the element's initial value is undefined.

Two symbols are specially recognized by the constructor:

:make-array Should be used only for **:array** and **:array-leader** type structures, or the named versions of those types.

:times Should be used only for **:grouped-array** type structures.

If one of these symbols appears instead of a slot name, it is interpreted just as the **:make-array** option or the **:times** option, and it overrides what was requested in that option.

For example:

```
(make-ship ship-x-position 10.0
           ship-y-position 12.0
           :make-array (:leader-length 5 :area disaster-area))
```

The order of evaluation of the initialization forms is not necessarily the same as the order in which they appear in the constructor call, nor the order in which they appear in the **defstruct**. You should make sure your code does not depend on the order of evaluation.

The *forms* are reevaluated every time a constructor is called. For example, if the form **(gensym)** is used as an initialization form (either in a call to a constructor or as a default initialization in the **defstruct**) then every call to the constructor creates a new symbol.

By-position Constructors for defstruct Structures

Note that **defstruct** defines a constructor function, but **zl:defstruct** defines a constructor macro.

If the **:constructor** option is given as **(:constructor** *name arglist***)**, then instead of making a keyword-driven constructor, **defstruct** or **zl:defstruct** defines a constructor that takes arguments interpreted by their position rather than by a keyword. The *arglist* is used to describe what the arguments to the constructor will be. In the simplest case, something like **(:constructor make-foo (a b c))** defines **make-foo** to be a three-argument constructor whose arguments are used to initialize the slots named **a**, **b**, and **c**.

In addition, you can use the keywords **&optional**, **&rest**, and **&aux** in the argument list. They work as you might expect, but note the following:

```
(:constructor make-foo
        (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines **make-foo** to be a constructor of one or more arguments. The first argument is used to initialize the **a** slot. The second argument is used to initialize the **b** slot. If there is no second argument, the default value (if any) given in the body of the **defstruct** or **zl:defstruct** is used instead. The third argument is used to initialize the **c** slot. If there is no third argument, the symbol **sea** is used instead. Any arguments following the third argument are collected into a list and used to initialize the **d** slot. If there are three or fewer arguments, **nil** is placed in the **d** slot. The **e** slot *is not initialized*; its initial value is undefined. Finally, the **f** slot is initialized to contain the symbol **eff**.

The actions taken in the **b** and **e** cases were carefully chosen to allow you to specify all possible behaviors. Note that the **&aux** "variables" can be used to completely override the default initializations given in the body.

Note that you are allowed to give the **:constructor** option more than once, so that you can define several different constructors, each with a different syntax.

The following restrictions should also be noted:

- For **zl:defstruct**, these "function-style" constructors do not guarantee that their arguments will be evaluated in the order which you wrote them.

- You cannot specify the **:make-array** or **:times** information in this form of constructor.

Alterant Macros for defstruct Structures

A call to the alterant macro has the form:

(*name-of-alterant-macro object*
slot-name-1 form-1
slot-name-2 form-2
...)

Object is evaluated, and should return an object of the structure. Each *form* is evaluated, and the corresponding slot is changed to have the result as its new value. The slots are altered after all the *form*s are evaluated, so you can exchange the values of two slots, as follows:

```
(alter-ship enterprise
    ship-x-position (ship-y-position enterprise)
    ship-y-position (ship-x-position enterprise))
```

As with the constructor macro, the order of evaluation of the *form*s is undefined. Using the alterant macro can produce more efficient code than using consecutive **setf**s when you are altering two byte fields of the same object, or when you are using the **:but-first** option.

You can use alterant macros on structures whose accessors require additional arguments. Put the additional arguments before the list of slots and values, in the same order as required by the accessors.

Differences Between defstruct and zl:defstruct

defstruct and **zl:defstruct** provide a similar functionality. **defstruct** adheres to the Common Lisp standard, with several extensions that were derived from useful features of **zl:defstruct**. **zl:defstruct** is supported for compatibility with previous releases.

Most of the documentation on **defstruct** pertains equally well to **zl:defstruct**. (See the section "Structure Macros".) This section describes the differences between **defstruct** and **zl:defstruct**.

- Constructor Difference

  defstruct defines constructor functions, whereas **zl:defstruct** defines constructor macros.

  When using the constructor function for a **defstruct**-defined structure, you give keyword arguments with the same name as the slots, to initialize the slots.

  ```
  (cl:defstruct door2 knob-color)
  (make-door2 :knob-color 'red)  ;slot name in keyword package
  ```

  When using the constructor macro for a **zl:defstruct**-defined structure, you give the names of the slots as the arguments to initialize the slots.

  ```
  (zl:defstruct door1 knob-color)
  (make-door1 knob-color 'blue)    ;slot name alone
  ```

- Option Differences

Most of the options accepted by **defstruct** are also accepted by **zl:defstruct**. Some of the options that are accepted by both have a slightly different behavior when given to **defstruct** than when given to **zl:defstruct**. The option with the most notable differences is **:type**. These differences are explicitly stated in the documentation: See the section "Options for **defstruct**".

The **defstruct**-only options are: **:print-function** and **:constructor-make-array-keywords**. The **zl:defstruct**-only option is **:export**.

- Default Behavior Differences

defstruct and **zl:defstruct** behave differently when no options are given. The differences in default behavior are noted below.

defstruct Default Behavior:

° The structure is implemented as a named vector. This means that by default, the **:named** option is implied. However, if you supply the **:type** option, the **:named** option is no longer implied; you should specify **:named** if you want a named structure.

° The name of the structure becomes a valid type specifier for **typep**.

° Accessor functions are defined for each slot, named by the convention:

structure-slot

° No alterant is defined, but you can use **setf** with an accessor function to change a slot value, such as:

`(setf (`*accessor object*`)` *new-value*`)`

° A copier function is defined, named by the convention:

copy-*structure*

° If the **:type** option is not given, or the **:type** and **:named** options are both given, a predicate function is defined, named by the convention:

structure-**p**

However, if **:type** is given and **:named** is not given, no predicate function is defined.

zl:defstruct Default Behavior:

° The structure is implemented as an unnamed array.

° The name of the structure does not become a valid type specifier for **typep**.

- Accessor functions are defined for each slot, named by the convention:

  slot

- An alterant function is defined, named by the convention:

  alter-*structure*

- You can use **setf** with an accessor function to change a slot value.

  `(setf (`*accessor object*`)` *new-value*`)`

- No copier function is defined.

- No predicate function is defined.

Advanced Use of defstruct

Functions Related to defstruct Structures

This summary briefly describes the functions related to **defstruct** structures.

defstruct *options* &body *items*
: Defines a new aggregate data structure with named components.

zl:defstruct
: Defines a new aggregate data structure with named components.

describe-defstruct *x* &optional *defstruct-type*
: Prints out a description of a given instance of a structure, including the contents of each of its slots.

defstruct-define-type *type* &body *options*
: Teaches **defstruct** and **zl:defstruct** about new types that it can use to implement structures.

sys:print-cl-structure *object stream depth*
: Function intended for use in a **defstruct :print-function** option; enables you to respect ***print-escape***.

sys:cl-structure-printer *structure-name object stream depth*
: Macro intended for use in a **defstruct :print-function** option; enables you to respect ***print-escape***.

Using defstruct Byte Fields

The byte field feature of **defstruct** or **zl:defstruct** allows you to specify that several slots of your structure are bytes in an integer stored in one element of the structure. For example, consider the following structure:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  (area-code 617)
  exchange
  line-number)
```

Although this works correctly, it wastes space. Area codes and exchange numbers are always less than **1000**, and so both can fit into **10** bit fields when expressed as binary numbers. To tell **defstruct** or **zl:defstruct** to do so, you can change the structure definition to one of the following forms.

Using **defstruct**, the syntax is:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  ((line-number)
   (area-code 617 :byte (byte 10 10))
   (exchange 0 :byte (byte 10 0))))
```

Using **zl:defstruct**, the syntax is:

```
(zl:defstruct (phone-book-entry (:type :list))
   name
   address
   ((area-code (byte 10 10) 617)
    (exchange (byte 10 0))
    (line-number)))
```

The lists **(byte 10 10)** and **(byte 10 0)** are byte specifiers to be used with the functions **ldb** and **dpb**. The accessors, constructor, and alterant macros now operate as follows:

```
(setq pbe (make-phone-book-entry
            :name "Fred Derf"
            :address "259 Orchard St."
            :exchange 232
            :line-number 7788))

=> (list "Fred Derf" "259 Orchard St." (dpb 232 12 2322000) 17154)

(phone-book-entry-area-code pbe) => (LDB (BYTE 10 10) (NTH 2 FOO))

(alter-phone-book-entry pbe
   area-code ac
   exchange ex)
```

```
=> ((lambda (g0530)
       (setf (nth 2 g0530)
             (dpb ac 1212 (dpb ex 12 (nth 2 g0530)))))
     pbe)
```

Note that the alterant macro is optimized to read and write the second element of the list only once, even though you are altering two different byte fields within it. This is more efficient than using two **setf**s. Additional optimization by the alterant macro occurs if the byte specifiers in the **defstruct** slot descriptions are constants.

If the byte specifier is **nil**, the accessor is defined to be the usual kind that accesses the entire Lisp object, thus returning all the byte field components as a integer. These slots can have default initialization forms.

The byte specifier need not be a constant; you can use a variable (or any Lisp form). It is evaluated each time the slot is accessed. Of course, you do not ordinarily want the byte specifier to change between accesses.

Constructor macros initialize words divided into byte fields as if they were deposited in the following order:

1. Initializations for the entire word given in the **defstruct** or **zl:defstruct** form.

2. Initializations for the byte fields given in the **defstruct** or **zl:defstruct** form.

3. Initializations for the entire word given in the constructor macro form.

4. Initializations for the byte fields given in the constructor macro form.

Alterant macros work similarly: The modification for the entire Lisp object is done first, followed by modifications to specific byte fields. If any byte fields being initialized or altered overlap each other, the actions of the constructor and alterant macros are unpredictable.

Grouped Arrays

The grouped array feature allows you to store several instances of a structure side-by-side within an array. This feature is somewhat limited; it does not support the **:include** and **:named** options.

The accessor functions are defined to take an extra argument, which should be an integer that acts as the index into the array of where this instance of the structure starts. This index should normally be a multiple of the size of the structure. Note that the index is the first argument to the accessor function and the structure is the second argument, the opposite of what you might expect. This is because the structure is &optional if the **:default-pointer** option is used.

Note also that the "size" of the structure (for purposes of the **:size-symbol** and **:size-macro** options) is the number of elements in *one* instance of the structure;

the actual length of the array is the product of the size of the structure and the number of instances. The number of instances to be created by the constructor macro is given as the argument to the **:times** option to **defstruct** or **zl:defstruct**, or the **:times** keyword of the constructor macro.

Named Structures

Introduction to Named Structures

The *named structure* feature provides a very simple form of user-defined data type. Any array can be made a named structure using **zl:make-array-into-named-structure**. See the function **zl:make-array-into-named-structure**. Usually however, the **:named** option of **defstruct** is used to create named structures. See the section "**defstruct** Structures and **type-of**".

The principal advantages of a named structure are that it has a more informative printed representation than a normal array and that the **describe** function knows how to give a detailed description of it. (You do not have to use **describe-defstruct**, because **describe** can figure out the names of the structure's slots by looking at the named structure's name.) We recommend, therefore, that "system" data structures be implemented with named structures.

Note: Flavors offers another kind of user-defined data type, more advanced but less efficient when used only as a record structure: See the section "Flavors".

A named structure has an associated symbol called its "named structure symbol", that it represents the user-defined type of which the structure is an instance. The **type-of** function, applied to the named structure, returns this symbol. If the array has a leader, the symbol is found in element 1 of the leader; otherwise it is found in element 0 of the array.

Note: If a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be stored in any element of a numeric array.

If you call **typep** with two arguments, the first an instance of a named structure and the second its named structure symbol, it returns **t**. It also returns **t** if the second argument is the named structure symbol of a **:named defstruct** included (using the **:include** option), directly or indirectly, by the **defstruct** for this structure. For example, if the structure **astronaut** includes the structure **person**, and **person** is a named structure, then giving **typep** an instance of an **astronaut** as the first argument, and the symbol **person** as the second argument, returns **t**. This reflects the fact that an astronaut is, in fact, a person, as well as an astronaut.

Handler Functions for Named Structures

You can associate with a named structure a function that handles various operations that can be done on it. You can control both how the named structure is printed and what **describe** will do with it.

To provide such a handler function, make the function the **named-structure-invoke** property of the named structure symbol. The functions that know about named structures apply this handler function to several arguments. The first is a "keyword" symbol to identify the calling function, and the second is the named structure itself. The rest of the arguments passed depend on the caller; any named structure function should have a "&rest" parameter to absorb any extra arguments that might be passed. What the function is expected to do depends on the keyword it is passed as its first argument. The following keywords are defined:

:which-operations

Returns a list of the names of the operations handled by the function.

:print-self

The arguments are **:print-self**, the named structure, the stream to which to output, the current depth in list-structure, and **t** if slashification is enabled (**prin1** versus **princ**). The printed representation of the named structure should be output to the stream. If the named structure symbol is not defined as a function, or **:print-self** is not in its **:which-operations** list, the printer defaults to a reasonable printed representation. For example:

#<*named-structure-symbol octal-address*>

:describe

The arguments are **:describe** and the named structure. It should output a description of itself to ***standard-output***. If the named structure symbol is not defined as a function, or **:describe** is not in its **:which-operations** list, the describe system checks whether the named structure was created by using the **:named** option of **defstruct**; if so, the names and values of the structure's fields are enumerated.

Here is an example of a simple named-structure handler function. For this example to have any effect, the person **defstruct** used in this example must be modified to include the **:named** attribute.

```
(defselect ((:property person named-structure-invoke))
           (:print-self (person stream ignore slashify-p)
            (format stream
                    (if slashify-p "#<person ~a>" "~a")
                    (person-name person))))
```

This example causes a person structure to include its name in its printed representation; it also causes **princ** of a person to print just the name, with no "**#<**" syntax.

In this example, the **:which-operations** handler is automatically generated, as well as the handlers for **:operation-handled-p** and **:send-if-handles**.

Another way to write this handler is as follows:

```
(defselect ((:property person named-structure-invoke))
  (:print-self (person stream ignore slashify-p)
   (if slashify-p
       (si:printing-random-object (person stream :typep)
         (princ (person-name person) stream))
       (princ (person-name person) stream))))
```

This example uses the **sys:printing-random-object** special form, which is a more advanced way of printing **#< ... >**. It interacts with the **si:print-readably** variable and special form.

Functions That Operate on Named Structures

named-structure-p *structure*
Returns **nil** if the given object is not a named structure.

named-structure-symbol *named-structure*
Returns the named structure symbol of the given named structure.

named-structure-invoke *operation structure* &rest *args*
Calls the handler function of the named structure symbol.

Also refer to the **:named-structure-symbol** keyword to **make-array**.

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:make-array-into-named-structure
Turns the given array into a named structure.

Extensions to defstruct

This section describes the use of **defstruct-define-type**.

An Example of defstruct-define-type

defstruct-define-type works by examining a call to the macro. This is how the **:list** type of structure might have been defined:

```
(defstruct-define-type :list
        (:cons (initialization-list description keyword-options)
               :list
               `(list . ,initialization-list))
        (:ref (slot-number description argument)
              `(nth ,slot-number ,argument)))
```

This is the simplest possible form of **defstruct-define-type**. It provides **defstruct** with two Lisp forms: one for creating forms to construct instances of the structure, and one for creating forms to become the bodies of accessors for slots of the structure.

The keyword **:cons** is followed by a list of three variables that are bound while the constructor-creating form is evaluated. The first, **initialization-list**, is bound to a list of the initialization forms for the slots of the structure. The second, **description**, is bound to the **si:defstruct-description** structure for the structure. See the section "**defstruct** Internal Structures". For a description of the third variable, **keyword-options**, and the **:list** keyword: See the section "Options to **defstruct-define-type**".

The keyword **:ref** is followed by a list of three variables that are bound while the accessor-creating form is evaluated. The first, **slot-number**, is bound to the number of the slot that the new accessor should reference. The second, **description**, is bound to the **si:defstruct-description** structure for the structure. The third, **argument**, is bound to the form that was provided as the argument to the accessor.

defstruct-define-type *type* &body *options* *Macro*

Teaches **defstruct** and **zl:defstruct** about new types that it can use to implement structures.

The body of this function is shown in the following example:

```
(defstruct-define-type type
        option-1
        option-2
        ...)
```

where each *option* is either the symbolic name of an option or a list of the form (*option-name* . *rest*). See the section "Options to **defstruct-define-type**".

Different options interpret *rest* in different ways. The symbol *type* is given an **si:defstruct-type-description** property of a structure that describes the type completely.

For a table of related items: See the section "Functions Related to **defstruct** Structures".

Options to defstruct-define-type

The documentation in this section regarding **defstruct** also applies to **zl:defstruct**.

:cons

Supplies **defstruct** with the code necessary to cons up a form that constructs an instance of a structure of this type.

The **:cons** option has the syntax:

```
(:cons (inits description keywords) kind
       body)
```

body is some code that should construct and return a piece of code that constructs, initializes, and returns an instance of a structure of this type.

The symbol *inits* is bound to the information that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization:

- **:list** — *inits* is bound to a list of initializations, in the correct order, with **nil**s in uninitialized slots.
- **:alist** — *inits* is bound to an alist with pairs of the form (*slot-number* . *init-code*).

The symbol *description* is bound to the instance of the **si:defstruct-description** structure that **defstruct** maintains for this particular structure. See the section "**defstruct** Internal Structures". This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* is bound to an alist with pairs of the form (*keyword* . *value*), where each *keyword* was a keyword supplied to the constructor macro that was not the name of a slot, and *value* was the Lisp object that followed the keyword. This is how you can make your own special keywords, such as the existing **:make-array** and **:times** keywords. See the section "Constructors for **defstruct** Structures". You specify the list of acceptable keywords with the **:keywords** option.

It is an error not to supply the **:cons** option to **defstruct-define-type**.

:ref

Supplies **defstruct** with the code it needs to cons up a form that will reference an instance of a structure of this type.

The **:ref** option has the syntax:

```
(:ref (number description arg-1 ... arg-n)
      body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* is bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the **si:defstruct-slot-description** structure. See the section "**defstruct** Internal Structures".

The symbol *description* is bound to the instance of the **si:defstruct-description** structure that **defstruct** maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally, there should be only one of these. The *last* argument is the one that is defaulted by the **:default-pointer** option. See the section "Options for **defstruct**". **defstruct** checks that the user has supplied exactly *n* arguments to the accessor function before calling the reference consing code.

It is an error not to supply the **:ref** option to **defstruct-define-type**.

:overhead

Declares to **defstruct** that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of **:overhead** is **(:overhead** *n***)**, where *n* is a fixnum that says how many locations of overhead this type needs.

This number is used only by the **:size-macro** and **:size-symbol** options to **defstruct**. See the section "Options for **defstruct**".

:named

Controls the use of the **:named** option to **defstruct**. With no argument, the **:named** option means that this type is an acceptable "named structure". With an argument, as in **(:named** *type-name***)**, the symbol *type-name* should be the name of some other structure type that **defstruct** should use if someone asks for the named version of this type. (For example, in the definition of the **:list** type the **:named** option is used like this: **(:named :named-list)**.)

:keywords

Allows you to define additional constructor keywords for this type of structure. (The **:make-array** constructor keyword for structures of type **:array** is an example.) The syntax is: **(:keywords** *keyword-1* **...** *keyword-n***)**, where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist. See the section "Options to **defstruct-define-type**".

:defstruct

Allows you to to run some code and return some forms as part of the expansion of the **defstruct** macro.

The **:defstruct** option has the syntax:

```
(:defstruct (description)
            body)
```

body is a piece of code that runs whenever **defstruct** is expanding a **defstruct** form that defines a structure of this type.

The symbol *description* is bound to the instance of the **si:defstruct-description** structure that **defstruct** maintains for this particular structure.

The value returned by *body* should be a *list* of forms to be included with those that the **defstruct** expands into. Thus, if you only want to run some code at **defstruct**-expand time, and you do not actually want to output any additional code, then you should be careful to return **nil** from the code in this option.

:predicate

Specifies how to construct a **:predicate** option for **defstruct**. The syntax for the option is:

```
(:predicate (description name)
            body)
```

The variable *description* is bound to the **si:defstruct-description** structure maintained for the structure for which a predicate is generated. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code that is evaluated to return the defining form for the predicate.

```
(:predicate (description name)
            `(defun ,name (x)
               (and (frobbozp x)
                    (eq (frobbozref x 0)
                        ',(defstruct-description-name)))))
```

:copier

The **:copier** option specifies how to copy a particular type of structure for situations when it is necessary to provide a copying function other than the one that **defstruct** would generate.

```
(:copier (description name)
  `(fset-carefully ',name 'copy-frobboz))
```

The syntax for the option is:

```
(:copier (description name)
         body)
```

description is bound to an instance of the **si:defstruct-description** structure, *name* is bound to the symbol to be defined, and *body* is some code to evaluate to get the defining form.

defstruct Internal Structures

The documentation in this section regarding **defstruct** also applies to **zl:defstruct**.

If you want to write a program that examines structures and displays them the way **describe** and the Inspector do, your program will work by examining the in-

ternal structures used by **defstruct**. In addition to discussing these internal structures, this section also provides the information necessary to define your own structure types.

Whenever you use **defstruct** to define a new structure, it creates an instance of the **si:defstruct-description** structure. This structure can be found as the **si:defstruct-description** property of the name of the structure; it contains such useful information as the name of the structure, the number of slots in the structure, and so on.

The following example shows a simplified version of how **si:defstruct-description** structure is actually defined. **si:defstruct-description** is defined in the **system-internals** (or **si**) package and includes additional slots that are not shown in this example:

```
;;;simplified version of si:defstruct-description structure
(cl:defstruct (defstruct-description
                (:default-pointer description)
                (:conc-name defstruct-description-))
          name
          size
          property-alist
          slot-alist)
```

The **name** slot contains the symbol supplied by the user to be the name of the structure, such as **spaceship** or **phone-book-entry**.

The **size** slot contains the total number of locations in an instance of this kind of structure. This is *not* the same number as that obtained from the **:size-symbol** or **:size-macro** options to **defstruct**. A named structure, for example, usually uses up an extra location to store the name of the structure, so the **:size-macro** option gets a number one larger than that stored in the **defstruct** description.

The **property-alist** slot contains an alist with pairs of the form (*property-name* . *property*) containing properties placed there by the **:property** option to **defstruct** or by property names used as options to **defstruct**. See the section "Options for **defstruct**".

The **slot-alist** slot contains an alist of pairs of the form (*slot-name* . *slot-description*). A *slot-description* is an instance of the **si:defstruct-slot-description** structure. The **si:defstruct-slot-description** structure is defined something like this, also in the **si** package:

```
;;;simplified version of the actual implementation
(cl:defstruct (defstruct-slot-description
                (:default-pointer slot-description)
                (:conc-name defstruct-slot-description-))
          number
          ppss
          init-code
          ref-macro-name)
```

Note that this is a simplified version of the real definition and does not fully represent the complete implementation. The **number** slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with **0**, and continuing up to one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure. See the section "Options to **defstruct-define-type**".

The **ppss** slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, the **ppss** slot contains **nil**.

The **init-code** slot contains the initialization code supplied for this slot by the user in the **defstruct** form. If there is no initialization code for this slot, the init-code slot contains the symbol **si:%%defstruct-empty%%**.

The **ref-macro-name** slot contains the symbol that is defined as a macro or a subst that expands into a reference to this slot (that is, the name of the accessor function).

Symbolics CLOS

Overview of the CLOS Documentation

Symbolics CLOS comes with the following documentation:

Object-Oriented Programming in COMMON LISP
: This book, by Sonya E. Keene, presents CLOS and techniques of object-oriented programming in a tutorial style. It is useful for Lisp programmers who are new to CLOS, or who are new to object-oriented techniques. This book contains many examples of programming with CLOS. This book is not available online via Document Examiner.

"Overview of CLOS"
: This section introduces the concepts of CLOS and shows a simple example of using CLOS. See the section "Overview of CLOS".

"Symbolics CLOS"
: This is the reference documentation for the Symbolics implementation of CLOS. It documents the important mechanisms of CLOS, and every CLOS operator (function, macro, and so on) in a reference style. It does not provide a tutorial approach or many examples, because that approach is taken by the book *Object-Oriented Programming in COMMON LISP*. For Genera users, the reference documentation is available online via Document Examiner.

  This documentation is based on information covered in the "Common Lisp Object System Specification", which is the complete definition of the behavior of CLOS and which should be considered the primary source of information on CLOS until

the ANSI specification of Common Lisp itself is completed and available. The authors of the "Common Lisp Object System Specification" (Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon) have given Symbolics their permission to include and/or adapt information from the specification into this reference documentation.

The CLOS specification is not provided with Symbolics CLOS. If you are interested in the CLOS specification, its reference is:

"Common Lisp Object System Specification," X3J13 Document 88-002R, June 1988. Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon.

The CLOS specification is available in:

SIGPlan Notices (ISSN 0362-1340)
Volume 23
Special Issue -- September 1988

ISBN 0-89791-289-6

Price: Members $10.50, Nonmembers $14.00.
ACM Order Number: 548883

Additional copies may be ordered prepaid from:
ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

CLOS Classes and Instances

This section gives more detail on CLOS classes and instances than the section ""Introduction to CLOS"". This section covers the following information:

- How to access slots for reading and writing.
- How CLOS classes are named.
- Integration of CLOS classes and the Common Lisp type system, including the set of predefined classes corresponding to Common Lisp types.
- The default classes **clos:standard-object** and **t**.

Other sections of the documentation present information related to classes:

See the section "CLOS Inheritance".

See the section "Creating and Initializing CLOS Instances".

See the section "Redefining CLOS Classes and Instances".

Accessing Slots of CLOS Instances

There are several ways to access slots. This section discusses accessors, **clos:slot-value**, **clos:with-slots**, and **clos:with-accessors**.

Note that local slots and shared slots are accessed in the same ways, and when they are accessed, there is no way of knowing whether the slot is local or shared.

Calling CLOS Accessors

Probably the most common way to access a slot's value is to call a *reader* to read the value of a slot, and to call a *writer* to write the value of a slot. Readers and writers are called *accessors*. The **clos:defclass** macro has slot options that automatically create methods for readers and writers.

To get the value of a slot using a reader generic function:

(*reader instance*)

The convention is that writers are **setf** generic functions that work with a reader. The name of such a writer is a list such as (**future-common-lisp:setf** *symbol*), and the writer is called with the **setf** syntax. (The difference between **setf** and **future-common-lisp:setf** is discussed below.) However, writers can also be named by symbols and be called with ordinary function-calling syntax.

To write the value of a slot using a writer generic function:

```
;;; if the CLOS writer's name is (future-common-list:setf symbol)
(setf (symbol instance) new-value)

;;; if the CLOS writer's name is writer
(writer new-value instance)
```

Accessors are normal generic functions. The **clos:defclass** slot options automatically create primary methods for accessors. You can specialize an accessor by defining another method on it, such as a before-method or after-method, to perform additional work.

Difference Between future-common-lisp:setf and setf

These two macros expand the same, and calling **future-common-lisp:setf** has the same effect as calling **setf**.

Because the argument order in defining setf methods and generic functions is different in CLOS and Flavors, the two symbols **setf** and **future-common-lisp:setf** are used in function specs for setf generic functions, to indicate which argument order is being used. The Flavors lambda-lists have the *new-value* parameter last, preceded by other arguments. The CLOS lambda-lists have the *new-value* parameter first, followed by other arguments.

The **:accessor** option to **clos:defclass** creates a method for a generic function whose function specs is of the form: (**future-common-lisp:setf** *symbol*).

The **:writable-instance-variables** option to **defflavor** creates a method for a generic function whose function spec is of the form: (**setf** *symbol*).

For more information on the differences between **future-common-lisp:setf** and **setf**: See the macro **future-common-lisp:setf**.

Using clos:slot-value

Accessors are implemented by **clos:slot-value**, a function that reads the value of any slot. You can use **setf** with **clos:slot-value** to set the value of a slot.

```
(slot-value instance slot-name)
(setf (slot-value instance slot-name) new-value)
```

The following example shows how you can define accessor methods yourself, using **clos:slot-value**.

```
;;; a method for a reader named x-position
(defmethod x-position ((p point))
  (slot-value p 'x-position))

;;; a method for the writer (future-common-lisp:setf x-position)
(defmethod (future-common-list:setf x-position) (new-value (p point))
  (setf (slot-value p 'x-position) new-value))
```

Contrasting Accessors and clos:slot-value

Accessors are a more abstract interface to a slot's value. They are named functions that take an object as the argument, and return or set some information about the object. The caller need not know that the function is implemented by reading or writing the value of a slot. Also, the developer can specialize an accessor with methods that perform additional work whenever a slot is accessed; these methods are called when the accessor generic function is used.

The function **clos:slot-value** is a more primitive interface, which requires that the caller knows that the desired information is stored in a slot of a given name. When **clos:slot-value** is called, the slot is accessed directly, and no accessor methods are invoked.

Short-cut Syntaxes: clos:with-slots and clos:with-accessors

Sometimes the syntax of accessors and **clos:slot-value** becomes cumbersome, and you might prefer a briefer syntax for calling an accessor or calling **clos:slot-value**.

The **clos:with-slots** macro provides a short-cut syntax for **clos:slot-value**. It enables you to specify a variable that stands for a call to **clos:slot-value**. You can use that variable to read the slot, and use **setf** or **setq** to write the slot.

Similarly, the **clos:with-accessors** macro provides a short-cut syntax for calling an accessor. It enables you to specify a variable that stands for a call to the accessor.

The **clos:with-slots** and **clos:with-accessors** macros are available for syntactic convenience only; they have no additional semantics.

Primitive Underlying Short-cut Syntaxes: clos:symbol-macrolet

The underlying mechanism of both **clos:with-accessors** and **clos:with-slots** is the **clos:symbol-macrolet** macro, which enables you to substitute forms for variable names within a lexical scope. Users can call **clos:symbol-macrolet** to provide short-cut syntaxes for other forms.

CLOS Operators for Accessing Slots

clos:slot-value *object slot-name*
: Returns the value of a given slot. You can use **setf** with **clos:slot-value** to change the value of a slot.

clos:symbol-macrolet *bindings* &body *body*
: Provides the underlying mechanism for substituting forms for variable names within a lexical scope; both **clos:with-accessors** and **clos:with-slots** are implemented via **clos:symbol-macrolet**.

clos:with-accessors *slot-entries instance-form* &body *body*
: Creates a lexical environment in which accessors can be called as if they were variables.

clos:with-slots *slot-entries instance-form* &body *body*
: Creates a lexical environment in which slots can be accessed by using variables that cause **clos:slot-value** to be called.

Class Objects and Their Names

A class is represented by a *class object*. Any class defined by **clos:defclass** also has a name.

There are two associations between a class and its name. A class is said to have a *proper name* if both associations reflect the same name for a class. The associations are:

- A class object knows its name. To find the name of a class object, you can use **clos:class-name**, and you can use **setf** with it to set the name of a class object.

- A symbol knows the class associated with it. To find the class object associated with a symbol, you can use **clos:find-class**, and you can use **setf** with it to set the class object associated with the symbol.

When you define a class with **clos:defclass**, these associations are set up automatically, and the class has a proper name.

Since **setf** can be used with both **clos:class-name** and **clos:find-class**, it is possible for these associations to reflect different names for a class. In these cases, the

class does not have a proper name. Note that you can break the association between a symbol and a class by using **(setf (clos:find-class** *name*) **nil)**.

You can use **clos:class-of** to get the class of a given object.

In summary, the operators that handle class names and class objects are:

clos:class-name *class*
: Returns the name of the class object. You can use **setf** with **clos:class-name** to set the name of the class object.

clos:class-of *object*
: Returns the class of the given object. The returned value is a class object.

clos:find-class *class-name* &optional *(errorp* **t***) environment*
: Returns the class object named by *class-name* in the given *environment*. You can use **setf** with **clos:find-class** to change the class associated with the symbol *class-name*.

CLOS Classes and Types

CLOS classes are smoothly integrated into the Common Lisp type system. Every class name and class object is a type specifier. The **typep** and **subtypep** functions can be given a class name or class object as their type argument.

```
(typep object class-name)
(typep object class-object)
```

The forms above return true if the object is an instance of the class, or an instance of a subclass of the given class.

```
(subtypep class-name-1 class-name-2)
(subtypep class-object-1 class-object-2)
```

The forms above return true if the first class is a subclass of the second class, or if the classes are the same class.

Some Common Lisp types have corresponding classes, to enable users to define methods that specialize on those classes. The following table shows the classes that correspond to Common Lisp types of the same name.

Class	Class Precedence List
bignum	(bignum integer rational number t)
clos:array	(clos:array t)
bit-vector	(bit-vector vector clos:array sequence t)
character	(character t)
complex	(complex number t)
cons	(cons list sequence t)
double-float	(double-float float number t)
fixnum	(fixnum integer rational number t)
float	(float number t)
hashtable	(hashtable t)
integer	(integer rational number t)
list	(list sequence t)
null	(null symbol list sequence t)
number	(number t)
package	(package t)
readtable	(readtable t)
random-state	(random-state t)
ratio	(ratio rational number t)
rational	(rational number t)
sequence	(sequence t)
single-float	(single-float float number t)
string	(string vector clos:array sequence t)
symbol	(symbol t)
t	(t)
vector	(vector clos:array sequence t)

The classes named **bignum**, **double-float**, **fixnum**, and **single-float** are Symbolics CLOS extensions; these classes are not defined in the CLOS specification.

Note that the names of these classes are in the **cl** package except for the class representing arrays, which is in the **future-common-lisp** package. The reason for this is that **defstruct**-defined structures are of type **cl:array**, but not of type **array**.

This table includes all Common Lisp types that have corresponding classes. If a Common Lisp type does not appear in the table, then there is no corresponding class for it, and you may not define methods that specialize on that type.

Note that Symbolics CLOS currently does not support all the classes defined in the CLOS specification. The excluded classes are: **function**, **pathname**, and **stream**.

Note that the classes corresponding to Common Lisp types are defined solely for the purpose of enabling you to define methods that specialize on them. In other ways, these classes do not behave like user-defined classes: you cannot use **clos:make-instance** to create instances of them; you cannot redefine these classes, and so on.

In CLOS, **defstruct** defines a new class (as long as the **:type** option is not provided), and you may define methods that specialize on such a class. Like the class-

es corresponding to Common Lisp types, you cannot use **clos:make-instance** to create instances of them; you cannot redefine these classes, and so on. Lastly, you cannot use **clos:slot-value** on **defstruct**-defined classes.

CLOS Default Classes: clos:standard-object and t

The predefined class **clos:standard-object** is automatically included as a superclass of each user-defined class (a class defined by **clos:defclass**); it supports the default behavior of user-defined classes.

The predefined class **t** is automatically a superclass of every class except for the class **t** itself. The class **t** is a superclass of user-defined classes, predefined classes, classes defined by **defstruct** and any other classes; it appears as the last class in every class precedence list.

CLOS Inheritance

CLOS Class Precedence List

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 used to control inheritance of slots and slot options, and the precedence of methods during the generic dispatch procedure.

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.

No class appears more than once in a class precedence list.

CLOS computes the class precedence list based on the set of class definitions of the class and all its superclasses. The algorithm that CLOS uses ensures that the following rules are always obeyed:

Rule 1 Each class has precedence over its superclasses.

Rule 2 The precedence order of direct superclasses is controlled by the order in which they appear in the **clos:defclass** form. Each direct superclass has precedence over the classes that follow it in the list of direct superclasses in the **clos:defclass** form.

Thus, you as developer of a CLOS program set the constraints of the class precedence locally, in the **clos:defclass** forms.

Sometimes the class definitions set conflicting constraints, and there is no possible class precedence list that can be constructed according to the two rules. In this case, CLOS signals an error.

Sometimes when CLOS applies the two rules to the set of class definitions, the result is exactly one valid class precedence list. Often, however, some subsets of the classes have no constraints with respect to one another, and there could be several possible class precedence lists that obey the two rules. The algorithm used by CLOS chooses one of these to be the class precedence list. The algorithm is deterministic; it always produces the same class precedence list, based on a set of class definitions.

The general guideline for programmers is that the class definitions should reflect any precedence order dependencies of the program. If it is important that one superclass has precedence over another, then you can include them both as direct superclasses in the **clos:defclass** form in the correct order.

For a complete description of the algorithm CLOS uses, see the CLOS Specification.

Example 1

Given the following class definitions, we will compute the class precedence list for class **a**:

```
(defclass a (b c) ())
(defclass b (d e) ())
(defclass c (f g) ())
(defclass d () ())
(defclass e () ())
(defclass f () ())
(defclass g () ())
```

- The class definition of **a** states that **a** must precede **b** and **c** (Rule 1), and **b** must precede **c** (Rule 2).

- The class definition of **b** states that **b** must precede **d** and **e** (Rule 1), and **d** must precede **e** (Rule 2).

- The class definition of **c** states that **c** must precede **f** and **g** (Rule 1), and **f** must precede **g** (Rule 2).

The class precedence list for class **a** is:

```
(a b d e c f g clos:standard-object t)
```

Notice that **clos:standard-object** and **t** appear at the end of the class precedence list; this is true of all user-defined classes.

Notice that in this class precedence list that the class **b** and its superclasses appear together in the class precedence list followed by the class **c** and all its superclasses. This is one desired result of the algorithm CLOS uses; it attempts to keep a class and its superclasses together in the class precedence list, if this would not violate Rule 1 or 2.

Example 2

Given the following class definitions:

```
(defclass a (b c d) ())
(defclass b () ())
(defclass c () ())
(defclass d (c) ())
```

CLOS signals an error when trying to determine the class precedence list for class **a**, because there are conflicting constraints on the precedence of classes **d** and **c**:

- The class definition of **a** states that **c** must precede **d** (Rule 2).
- The class definition of **d** states that **d** must precede **c** (Rule 1).

Inheritance of Slots and clos:defclass Options

A class inherits slots, some slot options, and one class option from its superclasses.

Slots

In the simplest case, of all the classes in the class precedence list, no two classes define a slot with the same name. In this case, a class inherits all slots defined by its superclasses. The class inherits all local slots defined by its superclasses, in the sense that each instance stores its own value for the local slots. The class inherits all shared slots, in the sense that instances of the class can access the value of the shared slots.

In other cases, more than one class in the class precedence list specifies a slot with the same name. The slot is inherited, and its characteristics are determined by the slot options given by each class that specifies that slot according to the inheritance behavior of each slot option.

clos:defclass Slot Options

The inheritance behavior of each slot option is independent of the other slot options.

- The **:reader**, **:writer**, **:accessor**, and **:locator** slot options are not inherited. These slot options create methods, which are inherited in the sense that the methods are applicable for subclasses of the given class.
- The **:allocation** slot option is inherited. This slot option controls whether the slot is local or shared.

  If more than one class in the class precedence list defines a slot with the same name, then the allocation of the slot is controlled by the most specific class that

defines the slot of that name, whether or not the **:allocation** slot option was explicitly provided. Note that if a class defines a slot and the **:allocation** slot option is not provided, it is the same as if **:allocation :instance** were specified.

If the most specific class that defines the slot omits the **:allocation** slot option, or specifies **:allocation :instance**, then the class inherits a local slot. Each instance of the new class stores its own value for the slot.

If the most specific class that defines the slot specifies **:allocation :class**, then the class inherits the shared slot. Instances of the new class can access the value of the shared slot.

- The **:initform** slot option is inherited. The initform of a slot is provided by the most specific class in the class precedence list that supplies the **:initform** slot option.

- The **:initarg** slot option is inherited by union. A slot can have more than one initialization argument. All initialization arguments declared for a slot by any class in the class precedence list are valid initialization arguments for the slot.

- The **:type** slot option is inherited by intersection; that is, by **and**ing together all the type constraints provided by classes in the class precedence list that for a given slot. Thus, the type of a slot is declared to be of the type **(and** *type-1 type-2 type-3***)** if three classes in the class precedence list specified the types *type-1*, *type-2*, and *type-3* for the slot.

- The **:documentation** slot option is inherited. The documentation of a slot is provided by the most specific class in the class precedence list that supplies the **:documentation** slot option.

clos:defclass Class Options

The only class option that is inherited is **:default-initargs**. The set of default initialization arguments for a class is the union of the default initialization arguments provided by each class in the class precedence list. If more than one class provides a default value for a given initialization argument, then the default value is the one provided by the most specific class that provides a default value for that initialization argument.

CLOS Methods and Generic Functions

This section gives more detail on CLOS methods and generic functions than the section "Introduction to CLOS". This section covers the following information:

- Generic function objects and their names
- Method objects and a method's identity
- The CLOS generic dispatch procedure

- Lambda-list congruence rules for a generic function and its methods
- Keyword arguments in generic functions and methods

For additional information related to methods and generic functions:

See the section "CLOS Method Combination".

CLOS Generic Function Objects and Their Names

A CLOS generic function object has the following components:

- Set of methods
- Lambda-list
- Method-combination type
- Other information

A generic function object is a function, so it can be used as the first argument to **funcall** and **apply**.

Note that the name of a generic function can be a symbol or a list such as (**future-common-lisp:setf** *symbol*). If the name of the generic function is a symbol, the generic function is called in ordinary Lisp function-calling syntax. A generic function whose name is a list is a **setf** generic function, and it must be called with **setf** syntax as follows:

```
(setf (symbol object) new-value)
```

In CLOS, there are several ways to define a generic function. The most common way is to call **clos:defgeneric** or **clos:defmethod** to define a generic function with a *global name*. (Note that **clos:defmethod** defines a generic function with the given name, if it is not already defined.) The global naming of a generic function is analogous to the global naming of an ordinary function defined by **defun**; specifically, the function cell of a symbol stores the generic function object.

You can use **fdefinition** to get a generic function object whose name is either a symbol or a list. (Note that you can use **symbol-function** to get the generic function object named by a certain symbol, but it does not work on lists.)

```
(fdefinition symbol) => generic function object

(fdefinition '(future-common-lisp:setf symbol))
  => generic function object
```

Another way to define a generic function is to use one of the method-defining slot options to **clos:defclass**, such as **:reader**, **:writer**, and **:accessor**. These slot options define methods, and if the generic function is not defined, they define the generic function with the given global name.

Symbolics CLOS does not support the operators that define generic functions with *local names*. These operators are: **clos:generic-labels** (analogous to **labels**), **clos:generic-flet** (analogous to **flet**), and **clos:with-added-methods**.

Symbolics CLOS does not support the **clos:generic-function** macro, which defines an anonymous generic function.

CLOS Method Objects and Their Identities

Methods can be defined by **clos:defmethod**, the **:method** option to **clos:defgeneric**, and the method-defining slot options to **clos:defclass** (including **:reader**, **:writer**, and **:accessor**).

A CLOS method object contains the following information representing a method:

Method function
: Lisp code that is executed when the generic dispatch causes this method to be part of the implementation of a generic function call; the method function corresponds to the body of a **clos:defmethod**.

Specialized lambda-list
: An extension of an ordinary lambda-list in which any of the required parameters may 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 specialized parameter is a list in one of the following formats:

  (*variable-name class-name*)
  (*variable-name* `(eql` *form*`)`)

  One step in the generic dispatch procedure is to select the applicable methods. For information on this step: See the section "CLOS Generic Dispatch".

Parameter specializers
: A list of the parameter specializers; the order of these is significant. Each required parameter has a parameter specializer. The parameter specializer for any unspecialized parameter is the class named **t**.

  Note that CLOS distinguishes between a *parameter specializer name* (these appear in the **clos:defmethod** lambda-list) and the corresponding *parameter specializer object*. When the parameter specializer name is a class name, the corresponding object is the class object of that name. When the parameter specializer name is a list such as (**eql** *form*), the corresponding object is the list (**eql** *object*), where *object* is the result of evaluating *form*.

Method qualifiers
: A list of the non-**nil** atoms that are used by the method-combination type; note that the order of these atoms is significant.

Method objects are not named. It is more appropriate to consider the *identity* of a method. Consider that **clos:defmethod** defines a new method if that method does

not already exist, or redefines a method, if it does exist. Understanding a method's identity enables you to know whether a method exists with the same identity as the one being defined.

A method's identity is defined by the following:

- The generic function it implements
- Its parameter specializers (the parameter specializer objects and their order)
- Its method qualifiers (the atoms and their order)

You can use **clos:find-method** to get the method object of a method for a given generic function, with the given set of parameter specializers and method qualifiers.

See the section "Function Specs for CLOS Methods".

Symbolics CLOS currently does not support the operators that define methods for generic functions with *local names* or anonymous generic functions. These operators are: **clos:generic-labels**, **clos:generic-flet**, **clos:with-added-methods**, and **clos:generic-function**.

CLOS Generic Dispatch

This section describes a key mechanism of CLOS: the procedure by which CLOS chooses the correct implementation of a generic function based on the arguments to the generic function. The implementation consists of code that comes from one or more methods; this combined body of code is called the *effective method*.

We present the generic dispatch procedure in a conceptual way; note that Symbolics CLOS optimizes portions of the generic dispatch. The semantic effect is the same as what is documented here.

When a generic function is called, the CLOS generic dispatch procedure performs the following steps:

1. Finds the set of applicable methods.
2. Arranges the applicable methods in precedence order.
3. Uses the method-combination type of the generic function to determine how the applicable methods should be combined into the effective method.
4. Executes the effective method and returns its values.

Step 1: How CLOS 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. A required parameter can appear in the lambda-list as one of the following:

(*variable-name* `(eql` *form*`)`)
(*variable-name class-name*)
variable-name

When a parameter is specialized with (**eql** *form*), the *form* is evaluated once, at the time that the method is defined. If the value of *form* is *object*, then the argument satisfies the specialized parameter if the following form returns true:

`(eql` *argument 'object*)

When a parameter is specialized with a class name, the argument satisfies the specialized parameter if the argument is a member of the class; that is, if the following form returns true:

`(typep` *argument 'class-name*)

When a parameter is unspecialized (the *variable-name* appears as a lone symbol which is not enclosed within a list), any argument satisfies the parameter.

Step 2: How CLOS arranges the applicable methods in precedence order

The precedence order of methods is calculated based on the parameter specializers of the methods, the class precedence lists of the required arguments to the generic function, and the argument precedence order of the generic function. The result of this step is a list of applicable methods sorted from most specific to least specific.

CLOS starts with the set of the applicable methods determined in Step 1. Each method object has a list of parameter specializers derived from the lambda-list. (For a definition of parameter specializers, see the section "CLOS Method Objects and Their Identities".) An unspecialized parameter has the class named **t** as its parameter specializer.

CLOS compares the precedence of two methods by comparing pairs of parameter specializers. The argument precedence order of the generic function controls the order in which the parameter specializers are compared. The default argument precedence order is left-to-right. It can be changed by using the **:argument-precedence-order** option to **clos:defgeneric**.

- CLOS compares two methods by considering the first parameter specializer of each method. (Note that "first" is determined by the argument precedence order of the generic function; by default, it is the leftmost parameter specializer.)

- If the parameter specializers are not equivalent, then the methods are ordered on the basis of this pair of parameter specializers according to these rules:

  - A parameter specializer of (**eql** *object*) is more specific than a class.

  - If both parameter specializers are classes, then both classes appear in the class precedence list of the corresponding argument to the generic function. One class is more specific than the other in the class precedence list of the argument. Thus, the method whose parameter specializer is the more specific class has precedence over the other method.

- If the parameter specializers are equivalent, then the methods cannot be ordered on the basis of this pair of parameter specializers, so CLOS proceeds to consider the next parameter specializer of each method. The two methods are sorted according to the first pair of parameter specializers that are not equivalent.

- If all parameter specializers of two methods are equivalent, then the methods have different qualifiers. In this case, either of the two methods can be chosen to precede the other.

Step 3: How CLOS uses the method-combination type to construct the effective method

A method-combination type determines how applicable methods should be combined into an effective method. The method-combination type uses the sorted list of applicable methods as its input. It decides which methods should be executed, the order in which the methods should be executed, and the way that the values of the generic function are computed.

For information on how the default method-combination type constructs an effective method, see the section "The **clos:standard** Method Combination Type".

Step 4: How CLOS executes the effective method and returns its values

This step is straightforward, once Step 3 has constructed the effective method. CLOS invokes that body of code and returns its values as the values of the generic function.

CLOS Lambda-list Congruence Rules

In CLOS, the lambda-lists of a generic function and all its methods must be *congruent*. The generic function lambda-list sets a pattern, and all the methods must adhere to that pattern. If a method is defined that does not adhere to the pattern, then an error is signaled.

These rules define the congruence of the lambda-lists of a generic function and its methods:

- Each lambda-list must have the same number of required parameters.

- Each lambda-list must have the same number of optional parameters. Each method can supply its own default for an optional parameter, but the generic function may not supply defaults for optional parameters.

- If any lambda-list mentions **&rest** or **&key**, each lambda-list must mention one or both of them.

- If the generic function lambda-list mentions **&key**, each method must accept all of the keyword names mentioned after **&key**, either by accepting them explicitly, by specifying **&allow-other-keys**, or by specifying **&rest** but not **&key**. Each

method can accept additional keyword arguments of its own. The checking of the validity of keyword names is done in the generic function, not in each method. A method is invoked as if the keyword argument pair whose keyword is **:allow-other-keys** and whose value is **t** were supplied, though no such argument pair will be passed.

- The use of **&allow-other-keys** need not be consistent across lambda-lists. If **&allow-other-keys** is mentioned in the lambda-list of any applicable method or of the generic function, any keyword arguments may be mentioned in the call to the generic function.

- The use of **&aux** need not be consistent across methods.

If a method is defined before the generic function is defined, the generic function is automatically defined, and its lambda-list is derived from that of the method. If the lambda-list for the method mentions keyword arguments, the lambda-list of the generic function will mention **&key**, but will not name any keyword arguments.

Keyword Arguments in CLOS Generic Functions and Methods

When a generic function or any of its methods mentions **&key** in a lambda-list, the specific set of keyword arguments accepted by the generic function varies according to the applicable methods.

For a given generic function call, the set of keywords accepted includes:

- The keyword arguments accepted by all applicable methods

- Any keyword arguments mentioned after **&key** in the generic function definition

If the lambda-list of any applicable method or of the generic function definition contains **&allow-other-keys**, all keyword arguments are accepted by the generic function.

A method that has **&rest** but not **&key** does not affect the set of acceptable keyword arguments.

The lambda-list congruence rules require that each method accept all of the keyword arguments mentioned after **&key** in the generic function definition, by accepting them explicitly, by specifying **&allow-other-keys**, or by specifying **&rest** but not **&key**. Each method can accept additional keyword arguments of its own, in addition to the keyword arguments mentioned in the generic function definition.

If a generic function is passed a keyword argument that no applicable method accepts, an error is signaled.

For example, suppose there are two methods defined for **width** as follows:

```
(defmethod width ((c character-class) &key font) body)

(defmethod width ((p picture-class) &key pixel-size) body)
```

Assume that there are no other methods and no generic function definition for **width**. The evaluation of the following form will signal an error because the keyword argument **:pixel-size** is not accepted by the applicable method:

```
(width (make-instance 'character-class :char \#Q)
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form will signal an error, because the keyword argument **:glyph** is not accepted by the applicable method:

```
(width (make-instance 'picture-class :glyph (glyph \#Q))
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form will not signal an error if the class named **character-picture-class** is a subclass of both **picture-class** and **character-class**:

```
(width (make-instance 'character-picture-class :char \#Q)
       :font 'baskerville :pixel-size 10)
```

Redefining CLOS Generic Functions and Methods

CLOS enables you to redefine generic functions and methods dynamically.

Redefining a Generic Function

If you evaluate a **clos:defgeneric** form and the generic function already exists, the generic function is redefined. **clos:defgeneric** returns the modified generic function, which is **eq** to the original generic function. All methods on the generic function are preserved, except for those defined by the **:method** option to **clos:defgeneric**; a method defined by that option in the former **clos:defgeneric** but not the current **clos:defgeneric** form is removed from the generic function.

Redefining a Method

If you evaluate a **clos:defmethod** form and the method already exists, the method is redefined. The old method is replaced with the redefined method. Future calls to the generic function will use the redefined method.

For more information on determining whether the method already exists, see the section "CLOS Method Objects and Their Identities".

The lambda-list of the method must be congruent with the lambda-list of the generic function, or an error is signaled. See the section "CLOS Lambda-list Congruence Rules".

Removing a Method

You can remove a record from a generic function by calling **clos:remove-method**. Once the method is removed, it will not be used in any future calls to the generic function.

CLOS Method Combination

Each generic function has a method-combination type, which controls:

- Which method qualifiers are supported
- Which kinds of methods can use **clos:call-next-method** in their bodies
- The order in which the applicable methods are called
- How the values of the generic function are computed

CLOS enables you to define new method-combination types. For information and examples, see the macro **clos:define-method-combination**.

This section describes the default method-combination type (whose name is **clos:standard**) and the other built-in method-combination types. It also gives a table of operators related to defining new method-combination types.

The clos:standard Method Combination Type

The default method-combination type is **clos:standard**. Unless a generic function specifies a different method-combination type by using the **:method-combination** option to **clos:defgeneric**, then the generic function uses **clos:standard**.

The **clos:standard** method-combination type supports the following method qualifiers:

none — An unqualified method is called a *primary method*. Its role is to perform the bulk of the work of a generic function. A primary method can call **clos:call-next-method** to call the next most specific primary method. A primary method returns its values to its caller.

:before — A method whose sole qualifier is **:before** is called a *before-method*. Its role is to run before the primary method(s), to do preparatory or initialization work. A before-method cannot call **clos:call-next-method**. Any values returned by a before-method are ignored.

:after — A method whose sole qualifier is **:after** is called an *after-method*. Its role is to run after the primary method(s), to do clean-up work. An after-method cannot call **clos:call-next-method**. Any values returned by an after-method are ignored.

:around — A method whose sole qualifier is **:around** is called an *around-method*. Its role is to be "wrapped around" the execution of the other methods (primary, before, and after). An around-method can call **clos:call-next-method**. The "next method" is the next most specific around-method, if there is one, otherwise the "next method" is the set of before-methods, the most specific

primary method, and the set of after-methods. An around-method returns its values to its caller.

The **clos:standard** method-combination type constructs an effective method in the following way:

- If there are any around-methods, the most specific around-method is called. It supplies the value or values of the generic function.

- Inside the body of an around-method, **clos:call-next-method** can be used to call the next method. When the next method returns, the around-method can execute more code, perhaps based on the returned value or values.

- If an around-method invokes **clos:call-next-method**, the next most specific around-method is called, if one is applicable. If there are no around-methods or if **clos:call-next-method** is called by the least specific around-method, the other methods are called as follows:

  ° All the before-methods are called, in most-specific-first order. Their values are ignored.

  ° The most specific primary method is called. Inside the body of a primary method, **clos:call-next-method** may be used to call the next most specific primary method. When that method returns, the previous primary method can execute more code, perhaps based on the returned value or values. If **clos:call-next-method** is not used, only the most specific primary method is called.

  ° All the after-methods are called in most-specific-last order. Their values are ignored.

If no around-methods are applicable, the most specific primary method supplies the value or values returned by the generic function. If any around-methods are applicable, the most specific around-method supplies the value or values returned by the generic function.

An error is signaled if **clos:call-next-method** is used in a before-method or after-method.

The generic function **clos:no-next-method** is invoked if **clos:call-next-method** is used and there is no applicable method to call. The default method for **clos:no-next-method** signals an error.

Any method that can use **clos:call-next-method** can use the function **clos:next-method-p** to test whether a next method exists.

An error is signaled if there is an applicable method, but there is no applicable primary method.

CLOS Built-in Method-Combination Types

In addition to **clos:standard**, CLOS offers a set of built-in method combination types.

The built-in method-combination types (other than **clos:standard**) have the same semantics as method-combination types defined by the short form of **clos:define-method-combination**. We use the term "simple" method-combination type to refer to these types.

The simple built-in method combination types are:

`+`	`and`	`append`
`list`	`max`	`min`
`nconc`	`or`	`progn`

A simple method-combination type constructs an effective method by combining all applicable primary methods inside a call to a Lisp operator. For example, if there are no applicable around-methods, the **+** method-combination type constructs an effective method that looks like this:

```
(+ (primary-method-A generic-function-arguments)
   (primary-method-B generic-function-arguments)
   (primary-method-C generic-function-arguments))
```

Thus, the value of the generic function is the sum of the values of the applicable primary methods.

To specify that a generic function should use a method-combination type other than **clos:standard**, use the **:method-combination** option to **clos:defgeneric**. For example:

```
(defgeneric total-supply lambda-list
  (:method-combination +)
```

To define a primary method for use with a simple method-combination type, you must supply a symbol as the method qualifier; the symbol is the name of the method-combination type. For example, to define a primary method for use with the **+** method-combination type:

```
(defmethod total-supply + lambda-list body)
```

Each simple method-combination type supports two method roles:

Qualifier	*Role*
name of m-c type	Primary method, which is combined with other primary methods in a call to the operator.
:around	An around-method, which is wrapped around the call to the operator. If more than one around-method is applicalbe, the effective method calls the most specific around-method first. If that method calls **clos:call-next-method**, then the next most specific around-method is called. If there are no more around-methods, then the operator is called (and its arguments are the values of the applicable primary methods).

No other method qualifiers are accepted, so before-methods and after-methods (for example) cannot be used with simple method-combination types.

You can control the order in which the primary methods are combined in the call to the operator. The simple method-combination types take an *order* argument, which is either **:most-specific-first** or **:most-specific-last**. The default is **:most-specific-first**. The order argument is specified in the **:method-combination** option to **clos:defgeneric**, after the name of the method-combination type, as shown here:

```
(defgeneric boot-all-components lambda-list
  (:method-combination progn :most-specific-last)
```

CLOS Operators for Defining Method-Combination Types

The primary tool for defining new method-combination types is the **clos:define-method-combination** macro. Many of the related operators are used inside the call to **clos:define-method-combination**. For information and examples, see the macro **clos:define-method-combination**.

clos:call-method
: Used within effective method forms (forms returned by the body of **clos:define-method-combination**) to call a method.

clos:define-method-combination *name* &body *body*
: Defines a new method-combination type.

clos:invalid-method-error *method format-string* &rest *args*
: Used within method combination to signal an error when the method qualifiers of an applicable method are not valid for the method-combination type; it should be called only within the dynamic extent of a method-combination function.

clos:make-method
: A list such as (**#:make-method** *form*) can be used instead of a method object as the first subform of **clos:call-method** or as an element of the second subform of **clos:call-method**.

clos:method-combination-error *format-string* &rest *arguments*
: Used to signal an error within method combination; it should be called only within the dynamic extent of a method-combination function.

clos:method-qualifiers *method*
: Returns a list of the qualifiers of the *method*.

CLOS Procedural Definitions

In many situations, CLOS offers both a default behavior (which is expected to be appropriate for most programs) and a hook for overriding the default behavior. We use the term *procedural definition* to mean a situation in which CLOS guarantees that a certain procedure will be followed, including the calling of one or more generic functions which the user can specialize.

Controlling the Printed Representation and Description of Objects

For example, whenever the print system needs to produce a printed representation of an object, CLOS guarantees that the **clos:print-object** generic function is called. CLOS provides a default primary method for **clos:print-object**, but users can override that method by providing a more specific method. This enables you to control the printed representation of objects of a given class.

Similarly, whenever **describe** is called, CLOS guarantees that the **clos:describe-object** generic function is called. Users can override the default method with a more specific method.

The main reason for controlling the printed representation or description of objects is to shield clients of your program from seeing how the objects are implemented. Often the default method for **clos:describe-object** gives the names and values of all the slots; you might prefer to hide some of that information, or to present it differently. The way to do so is to provide a primary method which will override the system's default primary method.

clos:describe-object *object stream*
: Provides a mechanism for users to control what happens when **describe** is called for instances of a class. The default method lists the slot names and values.

clos:print-object *object stream*
: Provides a mechanism for users to control the printed representation of instances of a class. The default method uses the #<...> syntax.

Controlling Common Error Situations

CLOS provides a procedural definition for certain common error situations. When these situations happen, the default behavior is to signal an error, but you can override that behavior by providing a more specific primary method for the appropriate generic function.

The generic functions that can be specialized to control error situations are:

clos:no-applicable-method *generic-function* &rest *function-arguments*
: Provides a mechanism for users to control what happens when a generic function is called, and no method is applicable. The default method for **clos:no-applicable-method** signals an error.

clos:no-next-method *generic-function method* &rest *args*
: Provides a mechanism for users to control what happens when **clos:call-next-method** is called, and no next method exists. The default method for **clos:call-next-method** signals an error.

clos:slot-missing *class object slot-name operation* &optional *new-value*
: Provides a mechanism for users to control what happens when a slot's value is desired for access (when **clos:slot-value** is called, among other operations), and there is no slot of the given name accessible to the instance. The default method signals an error.

clos:slot-unbound *class instance slot-name*

Provides a mechanism for users to control what happens when a slot's value is desired for access and the slot is unbound. The default method signals an error.

Controlling Initialization

CLOS has four related initialization protocols (used in object creation, reinitialization of an instance, updating an instance because its class was changed, and updating an instance because its class was redefined), which have procedural definitions. In these procedural definitions, CLOS guarantees to call more than one generic function, and you can choose which one to specialize for your purpose. For more information:

See the section "Creating and Initializing CLOS Instances".
See the section "Redefining CLOS Classes and Instances".

Creating and Initializing CLOS Instances

Overview of Creating and Initializing CLOS Instances

You can create and initialize a new instance of a class by calling **clos:make-instance**.

clos:make-instance *class* &allow-other-keys

Creates, initializes, and returns a new instance of the given class.

The syntax of **clos:make-instance** is simple. The first argument is the class (either a class object or the name of a class), and the remaining arguments are *initialization arguments*.

The *initialization arguments* are alternating initialization argument names and values; they are used to initialize the new instance. Some initialization arguments fill slots with values, and others are used by *initialization methods* (user-defined methods that control some aspect of initialization).

The initialization protocol consists of the following mechanisms that enable you to control various aspects of the initialization:

- Declaring a symbol to be an initialization argument name for a slot. This is done by using the **:initarg** slot option to **clos:defclass**. When a slot has an associated initialization argument name, you can specify the initial value of the slot by providing the initialization argument name with a value in the call to **clos:make-instance**.

- Supplying a default value form for an initialization argument. This is done by using the **:default-initargs** class option to **clos:defclass**. The default value form

for an initialization argument is used if the initialization argument is not provided in the call to **clos:make-instance**. You can provide a default value form for any initialization argument, whether it is intended to fill a slot or to be used by an initialization method.

- Supplying a default value form for a slot; this is called an *initform*. This is done by using the **:initform** slot option to **clos:defclass**. A slot's initform is used if the slot is not initialized by an initialization argument. Thus, if no initialization argument for the slot is given in the call to **clos:make-instance** and if no initialization argument for the slot has a default value form, then the slot is initialized with its initform. (Note that a slot's initform is used in three contexts: when creating a new instance, when updating an instance because its class was redefined, and when updating an instance because its class was changed.) Defining initialization methods, which are methods for **clos:initialize-instance** or **clos:shared-initialize**. The **clos:initialize-instance** generic function is called only during instance creation, to initialize slots of a new instance. If you specialize **clos:initialize-instance** with after-methods, then the normal slot-filling behavior takes place, and then your method is executed. This is the most common way to control initialization with a method.

  The **clos:shared-initialize** generic function is called in four different contexts: when creating a new instance, when reinitializing an existing instance, when updating an instance because its class was redefined, and when updating an instance because its class was changed. You can specialize **clos:shared-initialize** to control what happens in all of those contexts.

  It is also possible to specialize **clos:make-instance**, because it is defined as a generic function; however, this is not often done in application programs.

The initialization protocol is powerful, and it offers you a wide range of functionality. Since several CLOS mechanisms require initialization, the **clos:make-instance** protocol overlaps with those related protocols. The four related initialization protocols are:

- Creation of a new instance
- Reinitialization of an existing instance
- Updating an instance because its class was redefined
- Updating an instance because its class was changed

The overlapping of these protocols enables you to develop code that affects only one of the protocols, or all of them. Sometimes it is useful to write a method that is called in all four cases, and CLOS enables you to do that with a method for **clos:shared-initialize**, instead of requiring you to define methods for four separate generic functions.

Note that the order of evaluation of default value forms for initialization arguments and the order of evaluation of **:initform** forms are undefined. If the order of evaluation is important, you should define an initialization method that performs the work, rather than using **:initform** and **:default-initargs**.

Initialization of a New CLOS Instance

This section describes the different steps that occur when a CLOS instance is created and initialized.

1. The user calls **clos:make-instance**. The first argument indicates the class. The remaining arguments are the initialization arguments.

2. **clos:make-instance** checks the validity of the initialization arguments. In brief, the valid initialization arguments include any initialization arguments declared by the **:initarg** slot option, any keyword arguments accepted by any applicable methods for **clos:initialize-instance** and **clos:shared-initialize**, and the keyword **:allow-other-keys**. For more information, see the section "Declaring Initargs for a Class".

3. **clos:make-instance** creates a new instance. By default, it is created in **sys:default-cons-area**, but the initialization argument **clos-internals:storage-area** can be used to specify another area.

4. **clos:make-instance** initializes the slots of the instance and runs user-defined initialization methods. It does this by calling the **clos:initialize-instance** generic function with the instance and the initialization arguments provided to **clos:make-instance** followed by the default initialization argument of the class.

   - The default primary method for **clos:initialize-instance** initializes the slots by calling the **clos:shared-initialize** generic function with the instance, **t**, and the initialization arguments provided to **clos:initialize-instance**. The effect is to initialize the slots as follows:

     ° For any initialization argument that is declared to fill a slot, if the initialization argument is given in the call to **clos:make-instance**, then the slot is initialized with that value.

     ° For any slots not initialized so far, the slot is initialized by the value of its default initialization argument, if there is one.

     ° For any slots not initialized so far, the slot is initialized by the value of its initform, if there is one.

     User-defined methods for **clos:initialize-instance** are executed in the normal way. The usual way for users to customize the initialization behavior is to specialize **clos:initialize-instance** by writing after-methods, which run after the slots are filled as described above.

   - When called by the default primary method for **clos:initialize-instance**, the default primary method for **clos:shared-initialize** initializes the slots as described above. User-defined methods for **clos:shared-initialize** are executed

in the normal way. The usual way for users to customize the initialization behavior is to specialize **clos:shared-initialize** by writing after-methods, which run after the slots are filled as described above.

5. **clos:make-instance** returns the initialized instance.

Any slot that is not initialized by these steps (including by a user-defined initialization method) is unbound. If you try to access an unbound slot, the **clos:slot-unbound** generic function is called; its default method signals an error.

For more information on initialization methods, see the section "Defining Initialization Methods for Object Creation".

Defining Initialization Methods for Object Creation

clos:make-instance is defined in a procedural way. It is defined to call **clos:initialize-instance** with a given set of arguments. The **clos:initialize-instance** generic function is defined to call **clos:shared-initialize** with a given set of arguments.

There are two ways to control the initialization of new instances with methods:

- Specialize **clos:initialize-instance**. This generic function is called only when a new instance is created, so user-defined methods are called only in this context.

- Specialize **clos:shared-initialize**. This generic function is called in four contexts, so user-defined methods are called in all four contexts:

  ° Creation of a new instance
  ° Reinitialization of an existing instance
  ° Updating an instance because its class was redefined
  ° Updating an instance because its class was changed

When specializing either of those generic functions, the usual way is to define after-methods. Note that a user-defined primary method would override the default method, and thus could prevent the usual slot-initialization behavior from happening.

Note that any keyword arguments accepted by an applicable method for **clos:initialize-instance** or **clos:shared-initialize** is automatically declared as a valid keyword argument for a call to **clos:make-instance**.

When defining a method for one of these generic functions, if the method doesn't use any keyword arguments, generally it should specify **&key** with no named keywords. This makes the method's lambda-list to be congruent with the generic function. For more information, see the section "CLOS Lambda-list Congruence Rules".

CLOS offers some operators for dealing with slots that are useful within initialization methods. See the section "CLOS Operators for Use in Initialization Methods".

Declaring Initargs for a Class

Four CLOS mechanisms use initialization arguments for initialization purposes, and they each check the validity of the initialization arguments. If an invalid initialization argument is detected, an error is signaled.

The validity of initialization arguments is checked in the following contexts:

- Creating a new instance
- Reinitializing an existing instance
- Updating an instance because its class was redefined
- Updating an instance because its class was changed

There are two ways to declare initialization arguments as valid:

- Using the **:initarg** slot option to **clos:defclass** to declare a symbol as an initialization argument name that is used to initialize the value of a slot. (A single initialization argument can initialize more than one slot, if the same symbol is used in the **:initarg** option given for more than one slot.) Note that this slot option is inherited by union, so any symbol declared as an initialization argument by the class itself or any of its superclasses is valid for the class. Initialization arguments declared in this way are valid in each of the four contexts.

- Defining an initialization method that uses keyword parameters; all keywords defined by applicable initialization methods are automatically declared as valid initialization arguments. Each of the initialization protocols calls different methods, so each protocol has its own set of methods that declare initialization arguments as valid for that context.

  Creating a new instance
  : Keyword arguments accepted by any applicable methods for **clos:initialize-instance** and **clos:shared-initialize** are valid in this context.

  Reinitializing an existing instance
  : Keyword arguments accepted by any applicable methods for **clos:reinitialize-instance** and **clos:shared-initialize** are valid in this context.

  Updating an instance because its class was redefined
  : Keyword arguments accepted by any applicable methods for **clos:update-instance-for-redefined-class** and **clos:shared-initialize** are valid in this context.

  Updating an instance because its class was changed
  : Keyword arguments accepted by any applicable methods for **clos:update-instance-for-changed-class** and **clos:shared-initialize** are valid in this context.

Finally, the keyword **:allow-other-keys** is a valid initialization argument in all four contexts. The default value for **:allow-other-keys** is **nil**. If you provide **t** as its value, then all keyword arguments are valid.

Defining Constructors in CLOS

A constructor is a function that makes an instance of a given class. The name of the function and the arguments are tailored for that class. For example, you might define a constructor named **make-ship** to create instances of the class **ship**. Constructors give users a more abstract interface than using **clos:make-instance**.

CLOS does not have any feature that automatically defines constructor functions. Thus, if you want your program to use constructors, you must define them yourself. For example:

```
(clos:defclass ship () ((name :initarg :name)
                        (captain :initarg :captain)))

(defun make-ship (ship-name ship-captain)
  (clos:make-instance 'ship :name ship-name
                      :captain ship-captain))
```

CLOS Operators for Creating and Initializing Instances

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

clos:initialize-instance *instance* &allow-other-keys
: Calls **clos:shared-initialize** to initialize the instance, and returns the initialized instance.

clos:shared-initialize *instance slot-names* &allow-other-keys
: Initializes the *instance* according to the *initargs*, then initializes any unbound slots in *slot-names* according to their initforms, and returns the initialized instance.

CLOS Operators for Use in Initialization Methods

It can be useful to call these functions in initialization methods, including methods for **clos:initialize-instance**, **clos:shared-initialize**, **clos:update-instance-for-different-class** **clos:update-instance-for-redefined-class**, and **clos:reinitialize-instance**.

clos:slot-boundp *object slot-name*
: Returns true if the given slot has a value, otherwise returns false.

clos:slot-exists-p *object slot-name*
: Returns true if the *object* has a slot named *slot-name*, otherwise returns false.

clos:slot-makunbound *object slot-name*
Makes the given slot unbound.

Redefining CLOS Classes and Instances

You can redefine classes dynamically, even when instances of the class already exist. CLOS updates the instances to the new class definition. CLOS also enables you to change the class of an existing instance to a new class. Again, CLOS updates the affected instance automatically. However, you can control the updating of instances by specializing generic functions intended for this purpose. This section describes:

- How to redefine a class
- How to change the class of an instance
- How to reinitialize an instance
- How to control the updating of instances in these contexts

Redefining a CLOS Class

When you evaluate a **clos:defclass** form for a class that already exists, the effect is to *redefine* the class. The modified class object is **eq** to the original class object.

How CLOS Adds and Removes Accessor Methods

The redefinition of a class can result in the addition or removal of methods for accessor generic functions. A method is removed if it was specified in the old class definition by the **:accessor**, **:reader**, or **:writer** options, but it is not specified in the new class definition. (Here, "removed" means the method is removed from the generic function, as if **clos:remove-method** were called.) Similarly, a method is added if it is specified in the new class definition by the **:accessor**, **:reader**, or **:writer** options, but it was not specified in the old class definition. (Here, "added" means the method is defined for the generic function.)

How CLOS Updates Instances

If instances of the class (or of subclasses) exist at the time the class is redefined, CLOS updates the instances to adhere to the modified class definition. An instance is updated at some point in time before a slot of the instance is next accessed, so you are protected against accessing an instance that has not yet been updated.

Redefining a class can change the slots of the instances. If a slot is defined in the old class definition but not in the new, the slot is removed from instances. If a slot is defined in the new class definition but not in the old, the slot is added to instances. Slots can also be changed from local to shared, or shared to local.

How CLOS Handles Slots' Values

Once CLOS modifies the structure of an instance to contain the slots corresponding to the new class definition, it calls **clos:update-instance-for-redefined-class** to take care of retaining slots' values where appropriate, and initializing added slots with values.

The following table shows how the default method for **clos:update-instance-for-redefined-class** handles a slot's values in the different cases:

Old Definition	New Definition	Slot's Value
local	local	retained
shared	shared	retained
local	shared	initialized by initform
shared	local	retained
undefined	local	initialized by initform
undefined*	shared	initialized by initform
local	undefined	discarded
shared	undefined	discarded

*Note that in the case of a slot undefined in the old definition that is shared in the new defintion, it is **clos:defclass** that initializes the slot from its initform, not the default method for **clos:update-instance-for-redefined-class**.

When a slot's value is "retained", this means that the value is the same after class redefinition as it was before. If such a slot was unbound before class redefinition, it is unbound afterwards as well.

The default method for **clos:update-instance-for-redefined-class** does its initialization work by calling the **clos:shared-initialize** generic function with the instance, a list of the newly added local slots, and the initialization arguments provided to **clos:update-instance-for-redefined-class**. The second argument indicates that only the newly added local slots are to be initialized from their initforms.

The default method for **clos:update-instance-for-redefined-class** also checks the validity of the initialization arguments, and signals an error if an invalid initialization argument name is detected. See the section "Declaring Initargs for a Class".

Controlling the Initialization of Instances at Class Redefinition

There are two ways to control the initialization of instances when a class is redefined:

- Specialize **clos:update-instance-for-redefined-class**. This generic function is called only when instances are updated because a class was redefined, so user-defined methods are called only in this context.

When **clos:update-instance-for-redefined-class** is called, it receives several arguments that are useful for methods. It receives a list of the added local slots, a list of the discarded local slots, and a property list containing the names and values of the discarded local slots, and of any slots that are changing from local to shared. The property list enables your methods to access the slots' values just before they are permanently discarded.

- Specialize **clos:shared-initialize**. This generic function is called in four contexts, so user-defined methods are called in all four contexts:

  - Creation of a new instance
  - Reinitialization of an existing instance
  - Updating an instance because its class was redefined
  - Updating an instance because its class was changed

When specializing either of these generic functions, the usual way is to define after-methods. Note that a user-defined primary method would override the default method, and thus could prevent the usual slot-initialization behavior from happening.

It is essential that you specialize these generic functions before redefining the class. Otherwise, some or all of the instances might be updated in the default way (without running your methods).

Note that any keyword argument accepted by an applicable method for **clos:update-instance-for-redefined-class** or **clos:shared-initialize** is automatically declared as a valid keyword argument.

CLOS offers some operators for dealing with slots that are useful within initialization methods. See the section "CLOS Operators for Use in Initialization Methods".

The **clos:make-instances-obsolete** generic function is called automatically when a class is redefined to trigger the updating of instances; it can also be called by users to trigger the updating process and to invoke **clos:update-instance-for-redefined-class**.

Changing the Class of a CLOS Instance

You can change the class of an existing instance by calling **clos:change-class**. The modified instance is **eq** to the original instance. In this section, "previous class" means the original class of the instance and "current class" means the new class of the instance.

How CLOS Updates Instances

CLOS updates the instance to adhere to the definition of the current class. This can change the slots of the instance. If a local slot is defined in the previous class but not in the current class, the slot is removed from the instance. If a local slot is defined in the current class but not in the previous, the slot is added to the instance.

How CLOS Handles Slots' Values

Once CLOS modifies the structure of an instance to contain the slots corresponding to the current class, it takes care of retaining slots' values where appropriate, and initializing added slots with values. The following table shows what happens to the slots in the different cases:

Previous Class	Current Class	Slot's Value
local	local	retained (1)
shared	shared	replaced (2)
local	shared	replaced (2)
shared	local	retained (1)
undefined	local	initialized by initform (3)
undefined	shared	replaced (2)
local	undefined	discarded
shared	undefined	inaccessible

(1) "Retained" means that the slot's value is the same before and after the class is changed. If such a slot was unbound before the class is changed, it is unbound afterwards as well.

(2) Any slot defined by the previous class that is a shared slot in the current class has its value "replaced". This means that the instance no longer accesses the slot defined by the previous class, but it now accesses the shared slot defined by the current class. Thus, the value of the shared slot in the current class stays the same, and is now accessible to the instance.

(3) **clos:change-class** initializes the newly added local slots from initforms by calling **clos:update-instance-for-different-class**. The default method for **clos:update-instance-for-different-class** does its initialization work by calling the **clos:shared-initialize** generic function with the instance, a list of the newly added local slots, and the initialization arguments provided to **clos:update-instance-for-different-class**. The second argument indicates that only the newly added local slots are to be initialized from their initforms.

The default method for **clos:update-instance-for-different-class** also checks the validity of the initialization arguments, and signals an error if an invalid initialization argument name is detected. See the section "Declaring Initargs for a Class".

Controlling the Initialization of an Instance Whose Class Was Changed

There are two ways to control the initialization of instances when the class of an instance is changed:

- Specialize **clos:update-instance-for-different-class**. This generic function is called only when an instance is updated because its class was changed, so user-defined methods are called only in this context.

This generic function receives an argument called *previous*, which is a copy of the instance before its class was changed. The purpose of this argument is to enable methods to access the old slot values. It also receives an argument called *current*, which is the instance whose class has been changed. With these two arguments, your methods can initialize slots based on the values of slots in the previous version of the instance.

- Specialize **clos:shared-initialize**. This generic function is called in four contexts, so user-defined methods are called in all four contexts:

  ° Creation of a new instance
  ° Reinitialization of an existing instance
  ° Updating an instance because its class was redefined
  ° Updating an instance because its class was changed

When specializing either of these generic functions, the usual way is to define after-methods. Note that a user-defined primary method would override the default method, and thus could prevent the usual slot-initialization behavior from happening.

It is essential that you specialize these generic functions before changing the class of the instance. Otherwise, the instance will be updated in the default way (without running your methods).

Note that any keyword argument accepted by an applicable method for **clos:update-instance-for-different-class** or **clos:shared-initialize** is automatically declared as a valid keyword argument.

CLOS offers some operators for dealing with slots that are useful within initialization methods. See the section "CLOS Operators for Use in Initialization Methods".

Reinitializing a CLOS Instance

You can reinitialize an existing instance by calling **clos:reinitialize-instance**.

The reinitialization process is used to change the values of an instance's slots according to initialization arguments. There is no modification of an instance's structure to add or delete slots. The initform of a slot is not used in this process.

The default method for **clos:reinitialize-instance** does its initialization work by calling the **clos:shared-initialize** generic function with the instance, an empty list, and the initialization arguments provided to **clos:reinitialize-instance**. The second argument indicates that no slots are to be initialized from their initforms.

The default method for **clos:reinitialize-instance** also checks the validity of the initialization arguments, and signals an error if an invalid initialization argument name is detected. See the section "Declaring Initargs for a Class".

Controlling the Reinitialization of an Instance

There are two ways to control the initialization of instances when an instance is reinitialized:

- Specialize **clos:reinitialize-instance**. This generic function is called only when an instance is reinitialized, so user-defined methods are called only in this context. This generic function receives the instance as a required argument, and initialization arguments as an &rest argument.

- Specialize **clos:shared-initialize**. This generic function is called in four contexts, so user-defined methods are called in all four contexts:

  ° Creating a new instance
  ° Reinitializing an existing instance
  ° Updating an instance because its class was redefined
  ° Updating an instance because its class was changed

When specializing either of these generic functions, the usual way is to define after-methods. Note that a user-defined primary method would override the default method.

Note that any keyword argument accepted by an applicable method for **clos:reinitialize-instance** or **clos:shared-initialize** is automatically declared as a valid keyword argument.

CLOS offers some operators for dealing with slots that are useful within initialization methods. See the section "CLOS Operators for Use in Initialization Methods".

CLOS Operators for Redefining CLOS Classes and Instances

clos:change-class *instance new-class*
: Changes the class of the existing *instance* to *new-class*, and returns the modified *instance*. The modified instance is **eq** to the original instance.

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

clos:make-instances-obsolete *class*
: Called automatically when a class is redefined to trigger the updating of instances; it can also be called by users to trigger the updating process and to invoke **clos:update-instance-for-redefined-class**.

clos:reinitialize-instance *instance* &allow-other-keys
: Reinitializes an existing *instance* according to *initargs* and returns the initialized instance.

clos:shared-initialize *instance slot-names* &allow-other-keys
: Initializes the *instance* according to the *initargs*, then initializes any unbound slots in *slot-names* according to their initforms, and returns the initialized instance.

clos:update-instance-for-redefined-class *instance added-slots discarded-slots property-list* &allow-other-keys
Provides a mechanism for users to specialize the behavior of updating instances when a class is redefined.

clos:update-instance-for-different-class *previous current* &allow-other-keys
Provides a mechanism for users to specialize the behavior of updating an instance when its class is changed by **clos:change-class**.

Implementation Notes on Symbolics CLOS

Integration Between CLOS and Flavors

Mixed use of Flavors and CLOS is not supported at present. That is, a class cannot inherit from a flavor; a flavor cannot inherit from a class; CLOS generic functions cannot be used with Flavors methods and instances; and Flavors generic functions cannot be used with CLOS methods and instances.

Therefore if you convert a program to CLOS, you must convert all uses of a given flavor to use a class instead, and all Flavors methods for a given generic function to be CLOS methods instead.

Genera offers an automatic tool for converting Flavors programs to CLOS. CLOE does not have such a tool. Genera users only: see the section "Flavors to CLOS Conversion".

CLOS Packages

The **clos** package contains CLOS symbols. There are two categories of symbols in the **clos** package:

- Symbols which are included as a standard part of the CLOS specification (this includes Chapters 1 and 2 of the specification, which is the programmer interface to CLOS). These symbols are exported from the **clos** package, documented, and fully supported by Symbolics.

- Symbols which implement the stable and portable portions of the meta-object protocol. We believe that these symbols will be present in any future meta-object protocol standard. These symbols are exported from the **clos** package and can be used by users, although they are neither documented nor fully supported. These symbols are not defined in the CLOS specification, but many of them are present in compatible form in other implementations of CLOS. All are compatible in Symbolics Cloe. These symbols might change in a future release to reflect ongoing standardization efforts.

We recommend that users do not use the internal and external symbols of the **clos-internals** package, which implement the portions of the meta-object protocol

which we believe are not defined well enough to be standardized. A future meta-object protocol might provide similar functionality in a different way. These symbols are not exported from **clos**, not documented, and not supported.

The **future-common-lisp** and **future-common-lisp-user** packages contain an incomplete, unsupported implementation of the future ANSI Common Lisp standard. The **future-common-lisp** package contains symbols in the evolving ANSI Common Lisp standard. It is the ANSI Common Lisp equivalent of the **cl** package, although many of the functions have not yet been modified for ANSI Common Lisp, and many are not yet defined. Since ANSI Common Lisp includes CLOS, **future-common-lisp** contains the symbols in the CLOS package that are part of the standard. The **future-common-lisp-user** package is the ANSI Common Lisp equivalent of the **user** package, except that **future-common-lisp-user** uses only the **future-common-lisp** package; it does not include any Symbolics extensions to the language.

If a symbol in the **future-common-lisp** package is documented, then it is supported. We do not encourage users to depend on any undocumented symbols in the **future-common-lisp** package.

We chose the home package of a symbol according to whether the symbol was part of CLOS (**clos** is the home package) or something used in connection with CLOS (**future-common-lisp** is the home package).

Differences Between ANSI CLOS and Symbolics CLOS

This section describes how the current Symbolics CLOS implementation varies from the CLOS specification, which is part of the draft ANSI specification for Common Lisp. The differences fall into two categories: Symbolics extensions to CLOS, and CLOS features or functions that Symbolics does not support.

Symbolics Extensions to CLOS

- The **:locator** slot option to **clos:defclass** is a Genera extension, but CLOE does not support it.

- The function **locf** can be used with **clos:slot-value** to get a locative to a slot. This is a Genera extension, but CLOE does not support it.

- Many of the "Metaobject Protocol" generic functions are implemented to some extent. Although undocumented and subject to change, some of these, particularly the ones exported from the **clos** package, may be of interest.

Incompatibilities with the CLOS Specification

- Classes for the types **function**, **pathname**, and **stream** are not supported.

- The following operators are not implemented: **clos:generic-labels**, **clos:generic-flet**, **clos:generic-function**, and **clos:with-added-methods**.

- Methods that require a lexical environment are not yet supported.
- **clos:describe-object** returns the object being described. This is a deliberate incompatibility.
- When using **clos:ensure-generic-function**, if the **:lambda-list** is not provided, the lambda list of the generic function will not be changed. This reflects a change that will be made to the CLOS specification.
- **clos:symbol-macrolet** does not allow declarations.
- Note that the definition of **clos:symbol-macrolet** has been changed by X3J13 since the CLOS specification was published; the macro expansions are in the environment of the **clos:symbol-macrolet**.

Function Specs for CLOS Methods

The function spec of a CLOS method is:

```
(clos:method generic-function specializers . qualifiers)
```

generic-function is a symbol naming the generic function.

specializers is a list which has one element corresponding to each required parameter of the method. The element corresponding to a parameter specialized on a class is the name of that class. The element corresponding to an unspecialized parameter is **t**. The element corresponding to a parameter such as (**eql** *form*) is the list that appeared in the **clos:defmethod** form. Thus, one example of *specializers* is:

```
(integer symbol (eql *foo*))
```

qualifiers is a list containing the method's qualifiers such as:

```
:after
```

If you are editing a method, you can use the function spec as described above, or (as in Flavors), you can use the following:

```
(generic-function specializers . qualifiers)
```

You can use these function specs to edit the definition of a method and to get the arglist. You cannot use them with **trace** or **breakon**. You cannot use them to define a method, but you can use them to undefine a method (although m-X Kill Definition would probably be the more convenient way to kill the definition of a method, and this works on CLOS methods).

Show CLOS Commands

Show Class Generic Functions **Command**

Show Class Generic Functions *class keywords*

Shows all generic functions applicable to instances of the given class.

keywords :Detailed, :Match, :More Processing, :Output Destination.

:Detailed {Yes, No} Shows arguments, indicating (with "<!>") those which can be specialized for this class. The default is No (mentioned default is Yes).

:Match {*string*} Shows only generic functions whose names contain this substring. The default is to show all generic functions.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Show Class Initargs **Command**

Show Class Initargs *class keywords*

Shows the initialization arguments accepted by **clos:make-instance** of this class, and any default initial values. Also shows slots affected by initargs.

keywords :Detailed, :Direct, :Match, :More Processing, :Output Destination, :Sort

:Detailed {Yes, No} Shows affected slots and defaults. When :Detailed is No, you see the initializations from an external perspective (useful for making an instance). When :Detailed is Yes, you see the initializations from an internal perspective and get more information about how the class is constructed internally. (The mentioned default is Yes.)

:Direct {Yes, No} Shows only initializations defined in this class. With :Direct Yes, inherited initializations are not shown. The default

is No, which requests all initializations defined for this class or inherited by this class. (The mentioned default is Yes.)

:Match {*string*} Shows initargs whose names contain this substring. The default is to show all initargs.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Sort (Alphabetical, Class} Sorts the display alphabetically by initarg, or by location in the class precedence list. The default is Alphabetical. Class displays initargs in a format similar to the "Show Class Superclasses Command".

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Show Class Methods **Command**

Show Class Methods *class keywords*

Shows methods applicable to this class.

keywords :Direct, :Match, :More Processing, :Output Destination, :Stop At

:Direct {Yes, No} Shows only methods applicable to instances of the given class; does not show methods inherited from superclasses. The default is No (the mentioned default is Yes).

:Match {*string*} Shows only methods for generic functions whose names contain this substring. The default is to show methods for all generic functions.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** pro-

cessing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Stop At (*superclass-name*} Only look as far upward as this superclass, which is a superclass of the argument *class* (press HELP for a list of classes in their precedence order). The default is the direct subclass of **t** from which *class* inherits (the second-to-last element of *class's* precedence list).

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click M-Left to edit a definition.

Show Class Slots **Command**

Show Class Slots *class keywords*

Shows the slots of the given class.

keywords :Detailed, :Direct, :Match, :More Processing, :Output Destination, :Sort

:Detailed {Yes, No} Shows more detail, such as the accessors and initargs for the slots, as well as their names. The default is No (the mentioned default is Yes).

:Direct {Yes, No} Shows only direct slots (that is, those defined for this class). Slots inherited from superclasses are not shown. The default is No (the mentioned default is Yes).

:Match {*string*} Shows slots whose names contain this substring.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Sort — {Alphabetical, Class} Sorts the display alphabetically by slot name, or by location in the class precedence list. If Class, each slot is displayed along with the class that provides it. The default is Alphabetical.

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Show Class Subclasses **Command**

Show Class Subclasses *class keywords*

Shows the subclasses of this class.

keywords — :Brief, :Detailed, :Duplicates, :Initargs, :Levels, :Match, :Methods, :More Processing, :Output Destination, :Slots

:Brief — {Yes, No} :Brief Yes just shows the names of the subclasses of the class. :Brief No also shows the structure. The default is No (the mentioned default is Yes).

:Detailed — {Yes, No} Shows more detail, such as the accessors and initargs for the slots, as well as their names. (This option is only useful with :Slots.) The default is No (the mentioned default is Yes).

:Duplicates — {Yes, No} Shows classes everywhere they occur in the hierarchy, not just in the place from which they derive their precedence. The default is No (the mentioned default is Yes).

:Initargs — {*string*} Shows initargs whose names contain this substring. This is useful, because the initarg is shown next to the class from which it is inherited. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no initargs are requested.

:Levels — {*integer*} Specifies the number of levels of subclasses to show. Ellipses (...) indicate subclasses of a class that are not shown because of a :Level cutoff. The default is to show all subclass levels.

:Match — {*string*} Shows only the subclasses which contain this substring.

:Methods — {*string*} Shows methods for generic functions whose names match this substring. This is useful, because the method is shown next to the class from which it is inherited. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no methods are requested.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Slots {*string*} Shows slots whose names contain this substring. This is useful, because the slot is shown next to the class from which it is inherited. If *string* is omitted, shows all of them. If the keyword itself is not supplied, no slots are requested.

A subclass is a class that inherits from this class (directly or indirectly). This information is useful in program development or debugging, to answer the question "Which classes will be affected if I change the definition of this class?"

The output is indented to clarify the precedence order of the classes. The structure of the output is the inverse of the output of Show Class Superclasses.

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Show Class Superclasses **Command**

Show Class Superclasses *class keywords*

Shows the superclasses of this class in precedence order, from most specific to least specific. You can use this command to identify from which superclass a class derives part of its behavior.

keywords :Brief, :Detailed, :Duplicates, :Initargs, :Match, :Methods, :More Processing :Output Destination, :Slots

:Brief {Yes, No} :Brief Yes just lists the names of the superclasses of the class. :Brief No shows superclasses in outline form. The default is No (the mentioned default is Yes).

:Detailed {Yes, No} Shows more detail, such as the accessors and initargs for the slots, as well as their names. (This option is only useful with :Slots.) The default is No (the mentioned default is Yes).

:Duplicates {Yes, No} Shows classes everywhere they occur in the hierarchy, not just in the place from which they derive their precedence. The default is No (the mentioned default is Yes).

:Initargs {*string*} Shows initargs whose names contain this substring. This is useful, because the initarg is shown next to the class from which it is inherited. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no initargs are requested.

:Match {*string*} Shows only the superclasses which contain this substring.

:Methods {*string*} Shows methods for generic functions whose names match this substring. This is useful, because the method is shown next to the class from which it is inherited. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no methods are requested.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Slots {*string*} Shows slots whose names contain this substring. This is useful, because the slot is shown next to the class from which it is inherited. If *string* is omitted, shows all of them. If the keyword itself is not supplied, no slots are requested.

With :Brief No (the default), the classes are ordered from top to bottom, where the top class is the most specific class. The indentation graphically represents which classes are direct superclasses of which other classes.

With :Duplicates Yes, bracketed classes are duplicates that are included as direct superclasses, but are not ordered in this position, because of some ordering constraint. They appear in another place in the display without brackets, in their actual order. All bracketed classes have an arrow beside them. A down-arrow indicates that this class's position in the ordering is later in the display. An up-arrow indicates that this class's position in the ordering is earlier in the display (these occurrences are infrequent).

You can review the class precedence list by reading all unbracketed superclasses from top to bottom, ignoring punctuation. If :Duplicates is No, this is all that is displayed.

For information on how the order is determined, see the section "CLOS Class Precedence List".

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Show CLOS Generic Function **Command**

Show CLOS Generic Function *generic-function-name keywords*

Shows information about the given generic function.

keywords	:Classes, :Methods, :More Processing, :Output Destination, :Sort, :Specialized
:Classes	{Yes, No, By Class} Lists classes that have methods defined for this generic function. The default is No (the mentioned default is Yes). By Class is like Yes, except that it presents the class objects themselves, rather than the names of the classes.
:Methods	{Yes, No, Detailed} Lists methods of this generic function. The default is No (mentioned default is Yes). Detailed shows more information.
:More Processing	{Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").
:Output Destination	{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.
:Sort	{Alphabetical, Heuristic} Sorts the methods by specializer names either alphabetically or heuristically. (Used with :Methods Yes.) :Sort Heuristic sorts the methods in a way which takes into account the argument precedence order, the class precedence order of the specializers, the alphabetical order of the class names, and the lengths of the method qualifier lists.
:Specialized	Only shows methods applicable to particular specializers (otherwise all methods are shown). For each required argument of the generic function which can be specialized, you are prompted for an object to use for that argument, or a class to use as

a specializer for that argument, or a wildcard to indicate that you want to see all methods, regardless of their specializers. (Used with :Methods Yes.)

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Show Effective Method **Command**

Show Effective Method *generic-function specializers keywords*

Lists the applicable methods of *generic-function* with the arguments described by *specializers*. For each required argument of the generic function which can be specialized, you are prompted for an object to use for that argument, or a class to use as a specializer for that argument.

keywords :Code, :More Processing, :Output Destination

:Code {Yes, No} Shows code resulting from method combination. The default is No (the mentioned default is Yes).

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

Note that classes and methods are presented in the output in a way that enables you to click on them to perform operations on them. For example, you can click m-Left to edit a definition.

Summary of Symbolics CLOS Operators

This section summarizes all functions, macros, generic functions, and classes provided with Symbolics CLOS.

Basic Use of CLOS

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.

Handling CLOS Objects, and Their Names and Types

clos:class-name *class*
Returns the name of the class object. You can use **setf** with **clos:class-name** to set the name of the class object.

clos:class-of *object*
Returns the class of the given object. The returned value is a class object.

clos:find-class *class-name* &optional *(errorp* **t***) environment*
Returns the class object named by *class-name* in the given *environment*. You can use **setf** with **clos:find-class** to change the class associated with the symbol *class-name*.

clos:find-method *generic-function method-qualifiers specializers* &optional *errorp*
Returns the method object that is identified by *generic-function*, *method-qualifiers*, and *specializers*.

clos:remove-method *generic-function method*
Removes a method from a generic function and returns the modified generic function.

Accessing Slots

clos:slot-value *object slot-name*
Returns the value of a given slot. You can use **setf** with **clos:slot-value** to change the value of a slot.

clos:symbol-macrolet *bindings* &body *body*
Provides the underlying mechanism for substituting forms for variable names within a lexical scope; both **clos:with-accessors** and **clos:with-slots** are implemented via **clos:symbol-macrolet**.

clos:with-accessors *slot-entries instance-form* &body *body*
Creates a lexical environment in which accessors can be called as if they were variables.

clos:with-slots *slot-entries instance-form* &body *body*
Creates a lexical environment in which slots can be accessed by using variables that cause **clos:slot-value** to be called.

Calling and Testing for the Next Method

The method-combination type in use determines which kinds of methods support **clos:call-next-method** and **clos:next-method-p**. The **clos:standard** method-combination type supports these functions in around-methods and primary methods.

clos:call-next-method
Used within a method body to call the "next method".

clos:next-method-p
Called within the body of a method to determine whether a next method exists; returns true if a next method exists, otherwise returns false.

Creating and Initializing Instances

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

clos:initialize-instance *instance* &allow-other-keys
Calls **clos:shared-initialize** to initialize the instance, and returns the initialized instance.

clos:shared-initialize *instance slot-names* &allow-other-keys
Initializes the *instance* according to the *initargs*, then initializes any unbound slots in *slot-names* according to their initforms, and returns the initialized instance.

Redefining Classes and Instances

clos:change-class *instance new-class*
Changes the class of the existing *instance* to *new-class*, and returns the modified *instance*. The modified instance is **eq** to the original instance.

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

clos:make-instances-obsolete *class*
Called automatically when a class is redefined to trigger the updating of instances; it can also be called by users to trigger the updating process and to invoke **clos:update-instance-for-redefined-class**.

clos:reinitialize-instance *instance* &allow-other-keys
Reinitializes an existing *instance* according to *initargs* and returns the initialized instance.

clos:shared-initialize *instance slot-names* &allow-other-keys
Initializes the *instance* according to the *initargs*, then initializes any unbound slots in *slot-names* according to their initforms, and returns the initialized instance.

clos:update-instance-for-redefined-class *instance added-slots discarded-slots property-list* &allow-other-keys
Provides a mechanism for users to specialize the behavior of updating instances when a class is redefined.

clos:update-instance-for-different-class *previous current* &allow-other-keys
Provides a mechanism for users to specialize the behavior of updating an instance when its class is changed by **clos:change-class**.

Dealing with Slots in Initialization Methods

It can be useful to call these functions in initialization methods, including methods for **clos:initialize-instance**, **clos:shared-initialize**, **clos:update-instance-for-different-class clos:update-instance-for-redefined-class**, and **clos:reinitialize-instance**.

clos:slot-boundp *object slot-name*
Returns true if the given slot has a value, otherwise returns false.

clos:slot-exists-p *object slot-name*
Returns true if the *object* has a slot named *slot-name*, otherwise returns false.

clos:slot-makunbound *object slot-name*
Makes the given slot unbound.

Defining New Method-Combination Types

The primary tool for defining new method-combination types is the **clos:define-method-combination** macro. Many of the related operators are used inside the call to **clos:define-method-combination**. For information and examples, see the macro **clos:define-method-combination**.

clos:call-method
Used within effective method forms (forms returned by the body of **clos:define-method-combination**) to call a method.

clos:define-method-combination *name* &body *body*
Defines a new method-combination type.

clos:invalid-method-error *method format-string* &rest *args*
Used within method combination to signal an error when the method qualifiers of an applicable method are not valid for the method-combination type; it should be called only within the dynamic extent of a method-combination function.

clos:make-method
A list such as (**#:make-method** *form*) can be used instead of a method object as the first subform of **clos:call-method** or as an element of the second subform of **clos:call-method**.

clos:method-combination-error *format-string* &rest *arguments*
Used to signal an error within method combination; it should be called only within the dynamic extent of a method-combination function.

clos:method-qualifiers *method*
Returns a list of the qualifiers of the *method*.

Generic Functions Intended to Be Specialized

clos:describe-object *object stream*
Provides a mechanism for users to control what happens when **describe** is called for instances of a class. The default method lists the slot names and values.

clos:initialize-instance *instance* &allow-other-keys
Calls **clos:shared-initialize** to initialize the instance, and returns the initialized instance.

clos:no-applicable-method *generic-function* &rest *function-arguments*
Provides a mechanism for users to control what happens when a generic function is called, and no method is applicable. The default method for **clos:no-applicable-method** signals an error.

clos:no-next-method *generic-function method* &rest *args*
Provides a mechanism for users to control what happens when **clos:call-next-method** is called, and no next method exists. The default method for **clos:call-next-method** signals an error.

clos:print-object *object stream*
Provides a mechanism for users to control the printed representation of instances of a class. The default method uses the #<...> syntax.

clos:reinitialize-instance *instance* &allow-other-keys
Reinitializes an existing *instance* according to *initargs* and returns the initialized instance.

clos:shared-initialize *instance slot-names* &allow-other-keys
Initializes the *instance* according to the *initargs*, then initializes any unbound slots in *slot-names* according to their initforms, and returns the initialized instance.

clos:slot-missing *class object slot-name operation* &optional *new-value*
Provides a mechanism for users to control what happens when a slot's value is desired for access (when **clos:slot-value** is called, among other operations), and there is no slot of the given name accessible to the instance. The default method signals an error.

clos:slot-unbound *class instance slot-name*
Provides a mechanism for users to control what happens when a slot's value is desired for access and the slot is unbound. The default method signals an error.

clos:update-instance-for-different-class *previous current* &allow-other-keys
Provides a mechanism for users to specialize the behavior of updating an instance when its class is changed by **clos:change-class**.

clos:update-instance-for-redefined-class *instance added-slots discarded-slots property-list* &allow-other-keys
Provides a mechanism for users to specialize the behavior of updating instances when a class is redefined.

Miscellaneous CLOS Operators

future-common-lisp:setf *reference value* &rest *more-pairs*

Expands the same as does **setf**. Calling **future-common-lisp:setf** has the same effect as calling **setf**. Because the syntax for defining setf methods is different in CLOS and Flavors, these two symbols are used in function specs for setf generic functions, to indicate which syntax is being used.

future-common-lisp:defstruct *name-and-options* &body *slot-descriptions*

Defines a record-structure data type, and a corresponding class of the same name. You can define methods that specialize on structure classes. The syntax and semantics of **future-common-lisp:defstruct** adhere to the draft ANSI Common Lisp specification.

clos:make-load-form *object*

Provides a way to use an instance of a user-defined CLOS class (more precisely, an object whose metaclass is **clos:standard-class**) as a constant in a program compiled with **compile-file**. Users can define a method for **clos:make-load-form** that describes how an equivalent object can be reconstructed when the compiled-code file is loaded.

clos:make-load-form-saving-slots *object* &optional *save-slots*

This function can be useful in the bodies of methods for **clos:make-load-form**; it returns forms that construct an equivalent object, saving the values of some or all of its slots.

Functions Underlying the Commonly-Used Macros

These lower-level functions implement the commonly-used macros such as **clos:defmethod** and **clos:defgeneric**. They are not often needed in applications programs.

clos:add-method *generic-function method*

Adds *method* to *generic-function* and returns the modified *generic-function*; **clos:add-method** is the underlying mechanism of the **clos:defmethod** macro.

clos:compute-applicable-methods *generic-function arguments*

Returns the set of methods that are applicable for *function-arguments*; the methods are sorted according to precedence order.

clos:ensure-generic-function *function-name* &rest *keys* &key *:environment* &allow-other-keys

Defines a new generic function, or modifies an existing one. This function is part of the underlying implementation of **clos:defgeneric** and **clos:defmethod**.

clos:function-keywords *method*

Returns a list of the keyword parameter specifiers for *method* as its first value, and a boolean indicating whether **&allow-other-keys** was specified as its second value.

CLOS Classes and Meta-objects

These classes and meta-objects are predefined by CLOS. The predefined classes corresponding to Common Lisp types are listed elsewhere: See the section "CLOS Classes and Types".

clos:built-in-class
The class of many of the predefined classes corresponding to Common Lisp types, such as **list** and **t**.

clos:standard-class
The default class of classes defined by **clos:defclass**.

clos:standard-generic-function
The default class of generic function objects.

clos:standard-method
The default class of method objects.

clos:standard-object
This class is included implicitly as a superclass of every user-defined class; it provides default behavior for operations such as **clos:describe-object** and **clos:print-object**.

clos:structure-class
The class of classes defined by **defstruct**.

CLOS Operators Not Supported by Symbolics CLOS

clos:generic-flet *functions* &body *body*
Symbolics CLOS does not support **clos:generic-flet**.

clos:generic-function
Symbolics CLOS does not support **clos:generic-function**.

clos:generic-labels *functions* &body *body*
Symbolics CLOS does not support **clos:generic-labels**.

clos:with-added-methods
Symbolics CLOS does not support **clos:with-added-methods**.

Flavors

The documentation on Flavors is organized in four areas:

Overview:
"Overview of Flavors"

Basic Concepts:
"Basic Flavor Functions"
"Mixing Flavors"
"Example of Programming with Flavors: Life"

"Flavors Tools"
"Summary of Flavor Functions and Variables"

Advanced Concepts:
"Method Combination"
"Defining Functions Internal to Flavors"
"Wrappers and Whoppers"
"Complete Options for **defflavor**"
"Advanced Concepts for **defmethod**"
"Function Specs for Flavor Functions"
"Property List Methods"
"Generic Functions and Messages Supported by **flavor:vanilla**"
"Copying Instances"
"The Order of Defining Flavors and Flavor Operations"
"Implementation of Flavors"

Compatibility Features:
"Using Message-Passing Instead of Generic Functions"

We recommend that all users of Flavors read the sections noted as *Overview* and *Basic Concepts* above. Many Flavors-based programs do not require using any of the specialized programming practices described in *Advanced Concepts*.

If you do find that your program requires something extra, you can browse through the sections in *Advanced Concepts* to find the feature you are looking for.

The section in *Compatibility Features* describes message-passing, which is supported for compatibility with previous versions of Flavors. When writing new programs, it is good practice not to use message-passing. Because many system interfaces use message-passing, it is necessary to understand message-passing.

Basic Flavor Functions

This section describes several commonly used features, programming practices, and functions of Flavors.

Defining Flavors

The **defflavor** special form enables you to define flavors:

defflavor *name instance-variables component-flavors* &rest *options* *Special Form*

name is a symbol that is the name of this flavor.

defflavor defines the name of the flavor as a type name in both the Common Lisp and Zetalisp type systems; for further information, see the section "Flavor Instances and Types". **defflavor** also defines the name of the flavor as a presentation type name; for further information, see the section "User-defined Data Types as Presentation Types".

instance-variables is a list of the names of the instance variables containing the local state of this flavor. Each element of this list can be written in two ways: either the name of the instance variable by itself, or a list containing the name of the instance variable and a default initial value for it. Any default initial values given here are forms that are evaluated by **make-instance** if they are not overridden by explicit arguments to **make-instance**.

If you do not supply an initial value for an instance variable as an argument to **make-instance**, and there is no default initial value provided in the **defflavor** form, the value of an instance variable remains unbound. (Another way to provide a default is by using the **:default-init-plist** option to **defflavor**.)

component-flavors is a list of names of the component flavors from which this flavor is built.

Each *option* can be either a keyword symbol or a list of a keyword symbol and its arguments. The syntax of the **defflavor** *options* is given below, and the semantics of the options are described in detail elsewhere: See the section "Summary of **defflavor** Options". See the section "Complete Options for **defflavor**".

Several *options* affect instance variables, including:

:initable-instance-variables
:gettable-instance-variables
:locatable-instance-variables (not available in CLOE)
:readable-instance-variables
:settable-instance-variables
:special-instance-variables (not available in CLOE)
:writable-instance-variables

The options listed above can be given in two ways:

keyword
: The keyword appearing by itself indicates that the option applies to all instance variables listed at the top of this **defflavor** form.

(*keyword var1 var2 ...*)
: A list containing the keyword and one or more instance variables indicates that this option refers only to the instance variables listed here.

Briefly, the syntax of the other *options* is as follows:

:abstract-flavor
(**:area-keyword** *symbol*) (not available in CLOE)
(**:component-order** *args...*)
(**:conc-name** *symbol*)
(**:constructor** *args...*)
(**:default-handler** *function-name*)
(**:default-init-plist** *plist*)
(**:documentation** *string*)

(**:functions** *internal-function-names*)
(**:init-keywords** *symbols...*)
(**:method-combination** *symbol*)
(**:method-order** *generic-function-names*)
(**:mixture** *specs...*)
:no-vanilla-flavor (not available in CLOE)
(**:ordered-instance-variables** *symbols*)
(**:required-flavors** *flavor-names*)
(**:required-init-keywords** *init-keywords*)
(**:required-instance-variables** *symbols*)
(**:required-methods** *generic-function-names*)
(**:special-instance-variables-binding-methods** *generic-function-names*)
(not available in CLOE)

The following form defines a flavor **wink** to represent tiddly-winks. The instance variables **x** and **y** store the location of the wink. The default initial value of both **x** and **y** is **0**. The instance variable **color** has no default initial value. The options specify that all instance variables are **:initable-instance-variables**; **x** and **y** are **:writable-instance-variables**; and **color** is a **:readable-instance-variable**.

```
(defflavor wink ((x 0) (y 0) color)     ;x and y represent location
          ()                            ;no component flavors
  :initable-instance-variables
  (:writable-instance-variables x y)    ;this implies readable
  (:readable-instance-variables color))
```

You can specify that an option should alter the behavior of instance variables inherited from a component flavor. To do so, include those instance variables explicitly in the list of instance variables at the top of the **defflavor** form. In the following example, the variables **x** and **y** are explicitly included in this **defflavor** form, even though they are inherited from the component flavor, **wink**. These variables are made initable in the **defflavor** form for **big-wink**; they are made writable in the **defflavor** form for **wink**.

```
(defflavor big-wink (x y size)
          (wink)                        ;wink is a component
  (:initable-instance-variables x y))
```

If you specify a **defflavor** option for an instance variable that is not included in this **defflavor** form, an error is signalled. Flavors assumes you misspelled the name of the instance variable.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

Summary of defflavor Options

This section provides a brief summary of the options to **defflavor**. For a full description of each option, see the section "Complete Options for **defflavor**".

The following options are used frequently:

(**:initable-instance-variables** *vars...*)
Makes the specified instance variables *initable*, so you can initialize them when making an instance.

To specify all instance variables, give the symbol **:initable-instance-variables** alone, instead of using the syntax above.

(**:readable-instance-variables** *vars...*)
Makes the specified instance variables *readable*, by automatically creating an accessor function to read the value of each readable instance variable. If flavor *f* has a readable instance variable *v*, you can query an object for the value of *v* as follows:

(*f-v object*)

The accessor function is created in the current package.

To specify all instance variables, give the symbol **:readable-instance-variables** alone, instead of using the syntax above.

(**:writable-instance-variables** *vars...*)
Makes the specified instance variables *writable*. You can set the value of a writable instance variable by using **setf** and the accessor function (usually *f-v*):

(`setf` (*f-v object*) *value*)

All writable instance variables are also made readable.

To specify all instance variables, give the symbol **:writable-instance-variables** alone, instead of using the syntax above.

(**:locatable-instance-variables** *vars...*)
Enables you to get a locative pointer to the cell inside an instance that contains the value of an instance variable, using **locf** and the accessor function (usually *f-v*):

(`locf` (*f-v object*))

All locatable instance variables are also made readable.

To specify all instance variables, give the symbol **:locatable-instance-variables** alone, instead of using the syntax above.

This option is not supported by CLOE.

(**:conc-name** *symbol*)
Specifies a prefix for the accessor function created by the **:readable-instance-variables** option, which overrides the default (the name of the flavor followed by a hyphen).

(**:constructor** *args...*)
Generates a constructor function that creates new instances. These constructor functions are much faster than using **make-instance**. Constructor functions can accept positional arguments (instead of keyword arguments, like **make-instance**) or a mix of positional and keyword arguments. The constructor function is created in the current package.

(**:init-keywords** *symbols...*)
Declares its arguments as keywords that are accepted by **make-instance** of this flavor. Init keywords can be keyword arguments to be processed by methods defined for **make-instance** for this flavor, or keywords defined by the **:mixture** option to **defflavor**. See the section "Writing Methods for **make-instance**".

(**:default-init-plist** *plist*)
Provides a list of alternating initialization keywords and default value forms that are allowed for **make-instance** of this flavor. The keywords can be initable instance variables, init keywords, required init keywords, **:area**, or **:allow-other-keys**. You can specify any of these keywords as arguments to **make-instance** to override the default initial values given in the **:default-init-plist**.

(**:required-instance-variables** *symbols*)
Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables.

(**:required-init-keywords** *init-keywords*)
Specifies keywords that must be supplied (as an *init-option* to **make-instance** or in the **:default-init-plist** option to **defflavor**) when making an instance of this flavor.

(**:required-methods** *generic-function-names*)
Specifies generic functions that must be supported with methods by a flavor incorporating this one, if that flavor is intended to be instantiated.

(**:required-flavors** *flavor-names*)
Specifies flavors that must be included as components (directly or indirectly) by any flavor incorporating this one, if that flavor is intended to be instantiated.

(**:method-combination** *symbol*)
Declares the way that methods from different flavors are to be combined. See the section "Method Combination".

The following options are used less frequently:

(**:functions** *internal-function-names*)
Declares names of functions internal to the flavor. See the section "Defining Functions Internal to Flavors".

(**:mixture** *specs...*)
Defines a family of related flavors. When **make-instance** is called, it uses its keyword arguments (or defaults to values in the **:default-init-plist** of the **defflavor** form) to choose which flavor of the family to instantiate.

:abstract-flavor
Declares that this flavor is not intended to be instantiated; it is often used for the base flavor of a flavor family.

(**:component-order** *args...*)
Explicitly states the ordering constraints on the flavor components whose order is important.

(**:documentation** *string*)
Enables you to provide information on this flavor.

(**:area-keyword** *symbol*)
Changes the keyword that specifies the area in which the instance is made; this symbol can then be given to **make-instance**. This is useful if the flavor is using the **:area** keyword for some other purpose, such as an init keyword for an object's geometric or geographic area. Note that you cannot use this option with CLOE.

:no-vanilla-flavor
Specifies that **flavor:vanilla** should not be included in this flavor. This option is not supported by CLOE.

(**:ordered-instance-variables** *symbols*)
Specifies instance variables with fixed positions in the instance, for which speed is especially important. This increases efficiency at the cost of dynamic modification.

(**:method-order** *generic-function-names*)
Specifies names of generic functions for which speed is important. Flavors optimizes the speed for these functions.

The following options are available primarily for use by the system internals. When writing new programs, it is good practice not to use these options:

(**:special-instance-variables** *vars...*)
Specifies instance variables that should be bound as special variables when a particular method is called. This option is used in conjunction with **:special-instance-variable-binding-methods**. To specify all instance variables, give the symbol **:special-instance-variables** alone, instead of using the syntax above. This option is not supported by CLOE.

(**:special-instance-variables-binding-methods** *generic-function-names*)
Specifies certain methods that require the special instance variables to be bound as special variables. This option is used in conjunction with **:special-instance-variables**. This option is not supported by CLOE.

The following options are available for compatability with the previous flavor system. When writing new programs, it is good practice not to use these options:

(**:gettable-instance-variables** *vars...*)
Enables automatic generation of methods for getting the values of instance variables, via messages. To specify all instance variables, give the symbol **:gettable-instance-variables** alone, instead of using the syntax above. We suggest using the **:readable-instance-variables** options instead.

(**:settable-instance-variables** *vars...*)

Enables automatic generation of methods for setting the values of instance variables, via messages. To specify all instance variables, give the symbol **:settable-instance-variables** alone, instead of using the syntax above. We suggest using **:writable-instance-variables** and **:initable-instance-variables** instead.

(**:default-handler** *function-name*)

Specifies a function to be called when a generic function is called for which no method is available. This is the same as specifying a method for **:unclaimed-message**.

Flavor Instances and Types

defflavor defines the name of the flavor as a type name in both the Common Lisp and Zetalisp type systems. It is illegal for the name of a flavor to be the same as the name of an existing type (including a **defstruct** structure or a built-in type).

(type-of *instance***)** and **(:typep** *instance***)** return the symbol that is the name of the instance's flavor.

(typep *instance 'flavor-name***)** and **(typep** *instance 'flavor-name***)** return **t** if the flavor of *instance* is named *flavor-name* or contains that flavor as a direct or indirect component, **nil** otherwise.

Defining Methods

defmethod *Special Form*

A method is the code that performs a generic function on an instance of a particular flavor. It is defined by a form such as:

(`defmethod` *(generic-function flavor options...) (arg1 arg2...)*
body...)

The method defined by such a form performs the generic function named by *generic-function*, when that generic function is applied to an instance of the given *flavor*. (The name of the generic function should not be a keyword, unless you want to define a message to be used with the old **send** syntax.) You can include a documentation string and **declare** forms after the argument list and before the body.

A generic function is called as follows:

(*generic-function g-f-arg1 g-f-arg2...*)

Usually the flavor of *g-f-arg1* determines which method is called to perform the function. When the appropriate method is called, **self** is bound to the object itself (which was the first argument to the generic function). The arguments of the method are bound to any additional arguments given to the generic function. A method's argument list has the same syntax as in **defun**.

The *body* of a **defmethod** form behaves like the body of a **defun**, except that the lexical environment enables you to access instance variables by their names, and the instance by **self**.

For example, we can define a method for the generic function **list-position** that works on the flavor **wink**. **list-position** prints the representation of the object and returns a list of its x and y position.

```
(defmethod (list-position wink) () ; no args other than object
  "Returns a list of x and y position."
  (print self)                     ; self is bound to the instance
  (list x y))                      ; instance vars are accessible
```

The generic function **list-position** is now defined, with a method that implements it on instances of **wink**. We can use it as follows:

```
(list-position my-wink)
--> #<WINK 61311676>
=>  (4 0)
```

If no *options* are supplied, you are defining a primary method. Any *options* given are interpreted by the type of method combination declared with the **:method-combination** argument to either **defgeneric** or **defflavor**. See the section "Defining Special-Purpose Methods". For example, **:before** or **:after** can be supplied to indicate that this is a before-daemon or an after-daemon. For more information: See the section "Defining Before- and After-Daemons".

If the generic function has not already been defined by **defgeneric**, **defmethod** sets up a generic function with no special options. If you call **defgeneric** for the name *generic-function* later, the generic function is updated to include any new options specified in the **defgeneric** form.

Several other sections of the documentation contain information related to **defmethod**: See the section "**defmethod** Declarations". See the section "Writing Methods for **make-instance**". See the section "Function Specs for Flavor Functions". See the section "Setter and Locator Function Specs". See the section "Implicit Blocks for Methods". See the section "Variant Syntax of **defmethod**". See the section "Defining Methods to Be Called by Message-Passing".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

Defining Before- and After-Daemons

This section describes how to use the default type of method combination, which is **:daemon :most-specific-first**. This section introduces concepts of method combination and ordering of flavor components that are covered in detail elsewhere:

See the section "Inheritance of Methods".
See the section "Ordering Flavor Components".

Daemon is the most commonly-used method combination in the system, and it is the easiest to use. Daemon method combination type lets you use *before-daemons*

and *after-daemons*, which are two special types of methods. A before-daemon is a body of code that is to be executed before the primary method; an after-daemon is code to be executed after the primary method.

Typically, when you use **defmethod** to define a method to perform a generic function, you are defining a *primary* method. The primary method usually performs the main part of the generic function.

To write before-daemons and after-daemons, you write methods with the keywords **:before** or **:after** as the *options* argument in the **defmethod** form. You need not specify the **:method-combination** option to **defflavor** or **defgeneric**, because **:daemon** is the default method combination type.

For example, a program with a **rocket** flavor might have a generic function called **launch**. The primary method for **launch** simply launches the rocket. You might define a before-daemon that puts in the fuel, and an after-daemon that activates the radar tracking, as follows:

```
(defflavor rocket ((tank :empty) (position :ground))
           ()
  :initable-instance-variables)

(defmethod (launch rocket) ()                    ;primary method
  (setq position :space))

(defmethod (launch rocket :before) ()            ;before-daemon
  ;;; add fuel before launching
  (setq tank :full))

(defmethod (launch rocket :after) ()             ;after-daemon
  ;;; activate radar tracking after launch
  (setq *radar-tracking* t))
```

When you use the **launch** generic function on an instance of **rocket** flavor, the flavor system uses a *combined method* to perform the operation. The combined method resembles:

```
(flavor:multiple-value-prog2 (before-daemon)
                             (primary-method)
                             (after-daemon))
```

flavor:multiple-value-prog2 ensures that the generic function returns any values returned by its primary method.

In this example, one flavor supplied a before-daemon, a primary method, and an after-daemon. **rocket** has no component flavors, so there were no other methods to consider.

When flavors are built from components, often one flavor provides a primary method, and others provide before or after-daemons to be combined with that primary method. In **:daemon** method combination, all available before-daemons are run, a single primary method is run, and all available after-daemons are run.

Suppose **flav1** is built on two component flavors:

```
(defflavor flav1 () (flav2 flav3))
(defflavor flav2 () ())        ;flav2 has no components
(defflavor flav3 () ())        ;flav3 has no components
```

If all three flavors provide a before-daemon, a primary method, and an after-daemon for the same generic function, the combined method for performing that generic function on **flav1** resembles:

```
(flavor:multiple-value-prog2
   (progn (before-daemon-flav1)
          (before-daemon-flav2)
          (before-daemon-flav3))
   (primary-method-flav1)
   (progn (after-daemon-flav3)
          (after-daemon-flav2)
          (after-daemon-flav1)))
```

The order in which the methods are executed in the combined method depends on the ordering of flavor components. Flavors computes the ordering of flavor components for **flav1** as:

```
(flav1 flav2 flav3)
```

flav1 is said to be the *most-specific* flavor in the ordering, on the principle that specific flavors are usually built by including more general flavors as components. The **:daemon** method combination type states that before-daemons are executed in **:most-specific-first** method order, and after-daemons are executed in **:most-specific-last** method order. A single primary method is chosen from the set of available primary methods; it comes from the first flavor in the ordering that provides a primary method. For details on the ordering of flavor components, see the section "Ordering Flavor Components".

There can be only one before-daemon, one after-daemon, and one primary method written to implement a given generic function on any given flavor. If a second definition is given for an existing method, the previous method is overwritten and the new definition takes its place.

Flavors offers two advanced features that extend the flexibility of defining methods:

- Method Combination

  If the default way of method combination described here is not appropriate for your program, you can specify another type of method combination. See the section "Method Combination".

- Wrappers and Whoppers

  When before and after-daemons are not powerful enough, you can use wrappers and whoppers, which let you put some code *around* the execution of the method: See the section "Wrappers and Whoppers".

Compiling Flavor Methods

When you are developing flavors-based programs, **compile-flavor-methods** is a useful tool. It causes the combined methods of a program to be compiled at compile-time, and the data structures to be built at load-time, rather than both happening at run-time. **compile-flavor-methods** is thus a very good thing to use, since the need to invoke the compiler at run-time slows down a program using flavors the first time it is run. You use **compile-flavor-methods** by including it in a file to be compiled.

See the macro **compile-flavor-methods**.

Making Instances of Flavors

make-instance *flavor-name* &rest *init-options* *Function*

Creates and returns a new instance of the flavor named *flavor-name*, initialized according to *init-options*, which are alternating keywords and arguments. All *init-options* are passed to any methods defined for **make-instance**.

If **compile-flavor-methods** has not been done in advance, **make-instance** causes the combined methods of a program to be compiled, and the data structures to be generated. This is sometimes called *composing* the flavor. **make-instance** also checks that the requirements of the flavor are met. Requirements of the flavor are set up with these **defflavor** options: **:required-flavors**, **:required-methods**, **:required-init-keywords**, and **:required-instance-variables**.

init-options can include:

:initable-instance-variable *value*
: You can supply keyword arguments to **make-instance** that have the same name as any instance variables specified as **:initable-instance-variables** in the **defflavor** form. Each keyword must be followed by its initial value. This overrides any defaults given in **defflavor** forms.

:init-keyword *value*
: You can supply keyword arguments to **make-instance** that have the same name as any keywords specified as **:init-keywords** in the **defflavor** form. Each keyword must be followed by a value. This overrides any defaults given in **defflavor** forms.

:allow-other-keys t
: Specifies that unrecognized keyword arguments are to be ignored.

:allow-other-keys :return
: Specifies that a list of unrecognized keyword arguments are to be the second return value of **make-instance**. Otherwise only one value is returned, the new instance.

:area *number*

Specifies the area number in which the new instance is to be created. Note that you can use the **:area-keyword** option to **defflavor** to change the **:area** keyword to **make-instance** to a keyword of your choice, such as **:area-for-instances**.

Any ancillary values constructed by **make-instance** (other than the instance itself) are constructed in whatever area you specify for them; this is not affected by using the **:area** keyword. For example, if you supply a variable initialization that causes consing, that allocation is done in whatever area you specify for it, not in this area. For example:

```
(defflavor foo ((foo-1 (make-array 100)))
               ())
```

In this example the array is consed in **sys:default-cons-area**.

:area nil

Specifies that the new instance is to be created in the **sys:default-cons-area**. This is the default, unless the **:default-init-plist** option is used to specify a different default for **:area**.

If not supplied in the *init-options* argument to **make-instance**, the **:default-init-plist** option to the **defflavor** form is consulted for any default values for initable instance variables, init keywords, and the **:area** and **:allow-other-keys** options.

An alternative way to make instances is to use constructors. One advantage in using constructor functions is that they are much faster than using **make-instance**. You can define constructors by using the **:constructor** option; for more information, see the section "Complete Options for **defflavor**".

If you want to know what the allowed keyword arguments to **make-instance** are, use the Show Flavor Initializations command. See the section "Show Flavor Commands". `c-sh-A` works too, if the flavor name is constant.

You can define a method to run every time an instance of a certain flavor is created. For information, see the section "Writing Methods for **make-instance**".

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

Creates and returns a new instance of the flavor named *flavor-name*, initialized according to *init-options*, which are alternating keywords and arguments. All *init-options* are passed to any methods defined for **make-instance**.

If **compile-flavor-methods** has not been done in advance, **make-instance** causes the combined methods of a program to be compiled, and the data structures to be generated. This is sometimes called *composing* the flavor. **make-instance** also checks that the requirements of the flavor are met. Requirements of the flavor are set up with these **defflavor** options: **:required-flavors**, **:required-methods**, **:required-init-keywords**, and **:required-instance-variables**.

init-options can include:

:initable-instance-variable value
: You can supply keyword arguments to **make-instance** that have the same name as any instance variables specified as **:initable-instance-variables** in the **defflavor** form. Each keyword must be followed by its initial value. This overrides any defaults given in **defflavor** forms.

:init-keyword value
: You can supply keyword arguments to **make-instance** that have the same name as any keywords specified as **:init-keywords** in the **defflavor** form. Each keyword must be followed by a value. This overrides any defaults given in **defflavor** forms.

:allow-other-keys t
: Specifies that unrecognized keyword arguments are to be ignored.

:allow-other-keys :return
: Specifies that a list of unrecognized keyword arguments are to be the second return value of **make-instance**. Otherwise only one value is returned, the new instance.

If not supplied in the *init-options* argument to **make-instance**, the **:default-init-plist** option to the **defflavor** form is consulted for any default values for initable instance variables, init keywords, and the **:allow-other-keys** options.

If you want to know what the allowed keyword arguments to **make-instance** are, use the Show Flavor Initializations command.

You can define a method to run every time an instance of a certain flavor is created:

Writing Methods for make-instance

You can define a method to run every time an instance of a certain flavor is created, by defining a method for the generic function **make-instance**. This is a useful technique for performing additional initialization that depends on the instance variables being set to their initial values. If you write a method for **make-instance**, it is run after the instance is created and initialized according to *init-options*. Any values returned by **make-instance** methods are ignored.

Here is an example of a method defined for the **make-instance** function for the **freight-ship** flavor:

```
;;; Every time an instance of freight-ship is made,
;;; load extra fuel to move the ship faster, if:
;;;    cargo is perishable, and
;;;    we're in the hot season, and
;;;    the auxiliary fuel tank is empty.
(defmethod (make-instance freight-ship)
           (&key date-of-embarkation &allow-other-keys)
  (if (and (equal cargo-type :perishable)
           (hot-season-p date-of-embarkation)
           (equal auxiliary-fuel-tank :empty))
      (load-extra-fuel self)))
```

If a **make-instance** method allows an argument other than those that initialize instance variables, it is necessary to make that argument valid for **make-instance** of this flavor. To do so, use the **:init-keywords** option to **defflavor**. If the **make-instance** method requires any arguments, you should also use the **:required-init-keywords** option for **defflavor** to ensure that these arguments are always supplied. For example:

```
(defflavor freight-ship (cargo-type
                         primary-fuel-tank
                         auxiliary-fuel-tank)
           ()
  :initable-instance-variables
  (:init-keywords :date-of-embarkation)
  (:required-init-keywords :date-of-embarkation
                           :cargo-type
                           :auxiliary-fuel-tank))
```

make-instance methods receive all arguments that are given to **make-instance**. They also receive options not given to **make-instance** but present in the **:default-init-plist** option of the **defflavor** form (excluding options that initialize instance variables). Therefore, methods that use some of the arguments given to **make-instance** specify an argument list resembling:

(&key *arg1 arg2...* &allow-other-keys)

If you use &key, you must also use &allow-other-keys.

make-instance methods that do not use any of the arguments should specify **(&rest ignore)** as the argument list.

If a flavor has an **:init-keyword** named **foo** and an instance variable also named **foo**, you should use the more complex **&key** syntax:

```
(defmethod (make-instance f)
           (&key ((:foo foo-arg)) &allow-other-keys)
  body)
```

The generic function **make-instance** is implemented with the **:two-pass** method combination type. **:two-pass** means that the combined method executes all the primary methods and **:after** methods. **:after** methods are permitted because of unusual cases where a flavor's initializations must be performed after the initializations of its component flavors. **:before** methods are not allowed.

For any particular flavor, only one primary method and one **:after** method for **make-instance** are allowed. However, component flavors can supply primary methods and **:after** methods for **make-instance**; they are all run. When making an instance of a flavor that has component flavors, several of which have ordinary methods for **make-instance** and **:after** methods, the methods are executed as follows:

1. The instance is created and initialized according to *init-options*.

2. All primary methods for **make-instance** of this flavor and its component flavors are executed in most-specific-first order.

3. All **:after** methods for **make-instance** of this flavor and its component flavors are executed in most-specific-last order.

See the section "Built-in Types of Method Combination".

See the section "**:most-specific-first** and **:most-specific-last** Method Order".

Flavor System Compatibility

For compatibility with previous versions of the flavor system, an **:init** message is also sent if there are any methods for it. Normally, this practice should be avoided in new programs; the exception to that guideline is for window flavors.

make-instance methods are executed before **:init** methods. If you use **:after :init** methods for window flavors, we recommend that you continue to use them instead of replacing them with **make-instance** methods. The window system is not yet capable of handling **make-instance** methods well because it still uses **:init** methods internally. If you use **make-instance** methods it would be possible that your methods would be executed before the system's **:init** methods, which could lead to problems.

Generic Functions

There are two kinds of functions in Symbolics Lisp: ordinary functions and generic functions.

Ordinary Functions	*Generic Functions*
Have a single definition.	Have a distributed definition.

Interface is specified by **defun**.	Interface can be specified by **defgeneric** or **defmethod**.
Implementation is specified by **defun**.	Implementation is specified by one or more **defmethod** forms.
Do not treat flavor instances specially.	First argument is usually an instance of a flavor.
Implementation is the same whenever the function is called.	Implementation varies from call to call, depending on the flavor of the first argument to the function.

Ordinary functions and generic functions are called with identical syntax:

(*function-name arguments...*)

The first argument to a generic function is an object. The flavor of the object determines which method is invoked to perform the generic function.

Generic functions are not only syntactically compatible with ordinary functions; they are semantically compatible as well:

- The name of a generic function is in a certain package and can be exported if it is part of an external interface. This allows programmers to keep unrelated programs separate.
- They are true functions that can be passed as arguments and used as the first argument to **funcall** and **mapcar**. For example:

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

- Program development tools such as **trace** can be used on generic functions.

Usually, the generic function chooses the appropriate method by looking at the flavor of its first argument. You can specify alternate means of dispatching by **defgeneric** and **define-method-combination**.

Generic functions replace message passing used in the old flavor system. However, message passing is still supported for compatibility with old programs. See the section "Using Message-Passing Instead of Generic Functions".

Do not remove the property **flavor:generic** from a generic function; this causes internal problems in the Flavors system.

Use of defgeneric

It is not necessary to use **defgeneric** to write a generic function. If you define a method for an operation that has not been declared with **defgeneric**, **defmethod** automatically sets up a generic function with no special options.

defgeneric is used to:

- Formally define an interface.
- Specify options that pertain to the generic function as a whole.
- Gain the advantage of compile-time checking that the methods take the correct number of arguments, and that callers of the generic supply the correct number of arguments.
- Provide descriptive arguments to be displayed when you give the Arglist (m-X) command, or press c-sh-A for this generic function. (These default from the methods in a heuristic way if **defgeneric** is not used.)
- Document the contract of the generic function.

Here is an example of using **defgeneric** to document the contract of the generic function:

```
(defgeneric capacity (vehicle fare)            ;;; takes two args
  "Compute the number of passengers that can be
   accommodated at a given fare, and the number of
   crew required."
   (declare (values number-of-passengers
                    number-of-crew)))
```

You would use **capacity** as follows:

```
(capacity my-ship 17.50)
```

In the unusual case when you want to define a generic function whose name is a keyword, you must use **defgeneric**. If you try to give a keyword as a generic function name using **defmethod**, it is interpreted as a message name, which must then be called with **send**. Note that defining a function (generic or ordinary) whose name is a keyword is not a recommended practice.

defgeneric *name arglist* &body *options* *Special Form*

Defines a generic function named *name* that accepts arguments defined by *arglist*, a lambda-list. *arglist* is required unless the **:function** option is used to indicate otherwise. *arglist* represents the object that is supplied as the first argument to the generic function. The flavor of the first element of *arglist* determines which method is appropriate to perform this generic function on the object.

The semantics of the *options* for **defgeneric** are described elsewhere: See the section "Options for **defgeneric**". The syntax of the *options* is summarized here:

(**:compatible-message** *symbol*)
(**declare** *declaration*)
(**:dispatch** *flavor-name*)

(**:documentation** *string*)
(**:function** *body...*)
:inline-methods
(**:inline-methods :recursive**)
(**:method** (*flavor options...*) *body...*)
(**:method-arglist** *args...*)
(**:method-combination** *name args...*)
(**:optimize speed**)

For example, to define a generic function **total-fuel-supply** that works on instances of **army** and **navy**, and takes one argument (*fuel-type*) in addition to the object itself, we might supply military-group as *arg1*:

```
(defgeneric total-fuel-supply (military-group fuel-type)
  "Returns today's total supply
    of the given type of fuel
    available to the given military group."
  (:method-combination :sum))
```

The generic function is called as follows:

```
(total-fuel-supply blue-army ':gas)
```

The argument **blue-army** is known to be of flavor **army**. Therefore, Flavors chooses the method that implements the **total-fuel-supply** generic function on instances of the **army** flavor. That method takes only one argument, *fuel-type*:

```
(defmethod (total-fuel-supply army) (fuel-type)
  body of method)
```

The arguments to **defgeneric** are displayed when you give the Arglist (`m-X`) command or press `c-sh-A` while this generic function is current.

It is not necessary to use **defgeneric** to set up a generic function. For further discussion: See the section "Use of **defgeneric**".

The function spec of a generic function is described elsewhere: See the section "Function Specs for Flavor Functions".

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

Options for defgeneric

Each *option* is either a keyword symbol, a list whose car is a keyword symbol and whose cdr is arguments to the option, a **declare** form, or a documentation string. The following options are accepted:

"*documentation*" A string specifying self-documentation, for the same purpose as a **defun** documentation string. This is the same as giving the **:documentation** option.

(**:compatible-message** *symbol*)

Enables you to invoke the methods by either calling the *generic-function-name* or sending the *symbol* message. *symbol* is the name of the message; it is usually a keyword, but need not be. This option is for compatibility with old Flavors and will eventually be made obsolete. One example of the use of **:compatible-message** is in conjunction with a programming construct that recognizes messages only as selectors, such as **defselect**. For more information and an example of using this option: See the section "Defining a Compatible Message for a Generic Function".

(declare *declaration***)**

Declarations that apply to the whole function (as opposed to declarations of variables) are permitted. These include **arglist**, **values**, **sys:downward-funarg**, **sys:function-parent**, and **optimize**, among others. You can repeat this option any number of times.

(**:dispatch** *name*) Specifies the name of an argument whose flavor controls selection of methods. The arguments seen by the methods are the arguments to the generic function with *name* removed. Inside a method, **self** is bound to the object named by *name*. *name* is not mentioned explicitly in the **defmethod** arglist.

If **:dispatch** is not specified, the methods are selected on the basis of the flavor of the first argument to the generic function.

Dispatching off an argument other than the first argument is slightly slower because it involves an extra function call.

You can increase the efficiency by specifying **(:optimize speed)**. This gains performance at the cost of flexibility in program development. If you do this, you can no longer change anything about the generic function without having to recompile callers. See the section on the **:optimize** option for **defgeneric**.

(**:documentation** "*documentation*")

A string specifying self-documentation. This is the same as giving a documentation string.

(**:function** *body...*) Defines a function that runs instead of the generic dispatch. This is completely transparent to anyone calling the generic function. Such a prologue function can be used to rearrange the arguments, to standardize the arguments before the methods see them, to default optional arguments, to do the shared non-generic portion of an operation, or for any other purpose. To trigger the generic dispatch, apply the value of

(**flavor:generic** *generic-function-name*) to the arguments. See the special form **flavor:generic**. However, if the generic function uses the **:compatible-message** option, the way to trigger the generic dispatch is by sending the message, not by calling the generic function.

Here is an example of its use:

```
(defgeneric size-of-value (type value)
  (:function
    (if (eq value *null-value*) 0
        (funcall (flavor:generic size-of-value)
                 type value))))
```

An important use of this option is to extend a generic function so it can also be used for objects that are not flavor instances.

:inline-methods Indicates that performance of methods for this generic function is crucial. This prevents the method combination from generating function calls from combined methods to the methods they combined. Instead, the methods are coded inline. The implementation of these methods trades space for time.

(**:inline-methods :recursive**)
Causes even more inline coding to occur. That is, if a method that is being coded inline contains a form **(f self args...)**, and **f** is a generic function with the **(:inline-methods :recursive)** option, then the entire **f** operation is also coded inline. It is not necessary that **f** be the same generic function as the one for which this method is being defined.

We do not recommend that **:inline-methods** be used automatically for all generic functions. One example of an appropriate use of **:inline-methods** is in the Table Management facility. The methods for each type of table are assembled from a large number of pieces contributed by different mixin flavors. The use of **:inline-methods** enhances the performance of this program.

Note that inline coding happens only for combined methods where Flavors knows exactly what the flavor of **self** is. In the following example, **test** could not be compiled inline in this method, since there is no way of knowing that the method **(flavor:method baz foo)** will not be inherited by a flavor that also inherits a different method for **test** than **(flavor:method test foo)**.

```
(defgeneric test (object)
  (:inline-methods :recursive))
```

```
(defflavor foo () ())

(defmethod (test foo) () body)

(defmethod (baz foo) ()
  (test self))
```

(**:method** (*flavor options...*) *body...*)

This option lets you define a method for this generic function in the **defgeneric** form, instead of in a separate **defmethod** form. It is simply a convenient abbreviation for (**defmethod** (*generic-function flavor options...*) *arglist body...*) The argument list for the method is computed from the argument list given at the top of the **defgeneric** form. This option can be repeated any number of times.

As a matter of style, **:method** is often used to define a default method; this is done by defining a method for the most basic member of a flavor family. You can include a documentation string or **declare** forms before *body*.

(**:method-arglist** *args...*)

You can use this option to specify that the methods accept different arguments than does the generic function itself. By default, the methods receive the same arguments that are specified at the top of the **defgeneric** form, except for the dispatching argument. The **:method-arglist** option is only meaningful in connection with the **:function** option.

In the following example, **:method-arglist** is used to declare that the *x* argument is optional for the generic function but required for the methods. The generic function **gf1** does not require the object itself to be passed as an argument.

```
(defvar *obj*)
(defflavor fl1 () ())
(setq *obj* (make-instance 'fl1))

(defgeneric gf1 (&optional obj x)
  (:function
    (unless obj (setq obj *obj*))
    (funcall (flavor:generic gf1) obj x))
  (:method-arglist x))

(defmethod (gf1 fl1) (x)
  (format t "GF1 FL1 called on ~S ~S" self x))
```

For a slightly more complex example, we can reverse the order of the arguments:

```
(defgeneric gf1 (&optional x obj)
  (:function
    (unless obj (setq obj *obj*))
    (funcall (flavor:generic gf1) obj x))
  (:method-arglist x))
```

Doing **(gf1)** does the same thing as before, but now you can also do **(gf1 23)**.

(**:method-combination** *name args...*)
Specifies the type of method combination to be used in handling this generic function. If this option is used, all flavors must use the same method combination for this generic function. If the **:method-combination** option is also supplied to **defflavor**, that option must agree with the **:method-combination** option given to **defgeneric**. The default is **(:method-combination :daemon :most-specific-first)**. Note that this option is not available in CLOE.

For more information on usage and an example: See the section "Using the **:method-combination** Option".

(:optimize speed) Increases the speed of the generic function when dispatching off an argument other than the first argument. This has an effect only if **:dispatch** is also specified. The package of the symbol **speed** is unimportant; this syntax is consistent with the Common Lisp syntax for the **optimize** declaration.

If you use **(:optimize speed)** in conjunction with **:dispatch**, callers of the generic function are responsible for including the code to rearrange the arguments. The compiler puts in this code. Note that you must recompile all callers if you change anything about the generic function.

Redefining Flavors, Methods, and Generic Functions

You can redefine flavors, methods, and generic functions at any time. To do so, simply evaluate another **defflavor**, **defmethod**, or **defgeneric** form. The new definition replaces the old. This flexibility is useful in program development.

If you redefine a flavor, method, or generic function, the existing one is modified. The result is **eq** to the original one.

Redefining a Flavor

You can redefine a flavor by editing its **defflavor** form and then compiling the new definition, either by using the Zmacs command `c-sh-C`, or the **recompile-flavor** function.

Sometimes redefining a flavor causes old instances to be outdated; for example, adding or removing instance variables, or changing the order of instance variables. In these cases, Flavors gives you a warning that the flavor was changed in such a way that the stored representation is different. However, this does not cause a problem. When old instances are next accessed, they are updated to the new format. New instance variables will be initialized if the **defflavor** form indicates a default value, or left unbound otherwise. When a flavor is changed, the Flavors system propagates the changes to any flavors of which it is a direct or indirect component.

You can use **flavor:rename-instance-variable** to give an instance variable a new name, and to ensure that its value is preserved, for existing instances.

You can remove the definition of a flavor by using the Zmacs command Kill Definition (m-X), or the **flavor:remove-flavor** function.

Changing an Instance

You can explicitly change an existing instance in these ways: evaluate a new **defflavor** form; rename one or more of its instance variables using **flavor:rename-instance-variable**; or change the flavor of the instance using **change-instance-flavor**. If you redefine a flavor that has already been instantiated, this implicitly causes existing instances to be updated; this is described above. When you change instances, you should consider possible side effects; for example, any methods written for **make-instance** do not run when you change an instance. If you need to perform further initialization when an instance is changed, use **flavor:transform-instance**. See the generic function **flavor:transform-instance**.

Changing an instance in either of the ways described above modifies the original instance. The result is a modified instance which is **eq** to the original instance.

Redefining Generic Functions

Usually, you can redefine a generic function to be an ordinary function, or an ordinary function to be a generic function, without having to recompile any callers. However, if you use **defgeneric** and specify the **:dispatch** and **(:optimize speed)** options, you must recompile callers if you redefine the generic function.

Do not use **fundefine** to remove the definition of a generic function. If you do so, and then compile a **defmethod** form, the generic function remains undefined until you do an explicit **defgeneric**. While the generic function is undefined, any callers to it will malfunction. Also, do not remove the property **flavor:generic** from a generic function; this causes internal problems to Flavors.

Redefining Wrappers and Whoppers

Whoppers are functions, not macros, so they can be redefined at any time; the new definition replaces the old.

Redefining a wrapper automatically performs the necessary recompilation of the combined method of the flavor. If the wrapper is given a new definition, the combined method is recompiled so that it gets the new definition. If a wrapper is rede-

fined with the same old definition, the existing combined methods continue to be used, since they are still correct. The old and new definitions are compared using the function **equal**.

Because **defwhopper-subst** defines a wrapper, issues with redefining them are the same as for wrappers.

Related Functions:

change-instance-flavor *instance new-flavor*
Changes the flavor of an instance to another flavor.

recompile-flavor *flavor-name* &key *:generic :env :ignore-existing-methods (:do-dependents* **t***)*
Updates the internal data of the flavor and any flavors that depend on it.

flavor:remove-flavor *flavor-name*
Removes the definition of a flavor.

flavor:rename-instance-variable *flavor-name old new*
Changes the name of an instance variable, carrying the value of the old instance variable to the new for any existing instances.

flavor:transform-instance *instance new-fl*
Executes code when an instance is changed to a new flavor; thus enables you to perform initialization of the instance. Use this generic function by defining methods for it. It is not intended to be called.

Related Editor Tools:

See the section "Zmacs Commands for Flavors, Generic Functions, and Methods", for information on Zmacs tools to manipulate these constructs.

Mixing Flavors

It is advantageous to mix flavors when the characteristics of two or more different kinds of objects overlap. You identify the common characteristics and define a flavor that can be incorporated (or mixed) into both kinds of objects. This is called a *component flavor*.

For example, we might write a program involving space-ships and meteors. The characteristics of space-ships and meteors partially overlap; they both have x, y, and z-velocity. We can define a flavor called **3-d-moving-object** that represents their common characteristics:

```
(defflavor 3-d-moving-object (x-velocity y-velocity z-velocity)
           ()
  :initable-instance-variables)
```

Once **3-d-moving-object** is defined, we can include it in the definition of the flavors **space-ship** and **meteor** as follows:

```
(defflavor space-ship (crew-list name destination)
          (3-d-moving-object)          ;component flavor
  :initable-instance-variables)

(defflavor meteor (percent-iron estimated-mass)
          (3-d-moving-object)          ;component flavor
  :initable-instance-variables)
```

Objects of the **space-ship** and **meteor** flavors inherit the instance variables of their component flavor **3-d-moving-object**. When making an instance of **meteor** we can initialize its inherited variables:

```
(make-instance 'meteor :estimated-mass 27
                       :x-velocity 12
                       :y-velocity 44
                       :z-velocity 87)
```

When you use existing flavors to create a new flavor, the new flavor inherits characteristics of each of its component flavors (and of its components' components, and so on). This includes instance variables, methods, and other characteristics that are attached to a **defflavor** form, such as **:default-init-plist**.

Typically, a program has a family of related flavors. The most basic flavor has instance variables and methods defined that relate to the whole family of flavors. Flavors built on the basic flavor inherit those instance variables and methods, and include additional (more specialized) instance variables and methods. See the section "Flavor Families".

Inheritance of Methods

When a generic function is applied to an object of a particular flavor, methods for that generic function attached to that flavor or to its components are available. From this set of available methods, one or more are selected to be called. If more than one is selected, they must be called in some particular order and the values they return must be combined somehow.

Flavors constructs a single *handler* for each generic function supported by the new flavor. The handler is the code that actually performs the generic function on an instance of the flavor. In the simplest case, the handler is simply one of the methods.

Example of a Combined Method

In other cases the handler is a *combined method*. One example of a combined method is the handler constructed when the default method combination type (**:daemon**) is used, and a single **:before** method and **:after** method is present. The body of the combined method resembles:

```
(flavor:multiple-value-prog2 (before-method)
                             (primary-method)
                             (after-method))
```

Definition of Method Combination

The way that Flavors constructs a handler is called *method combination*. Often many methods are defined for performing a generic function on objects of a given flavor (some of the methods are defined for component flavors), and these methods must somehow be combined. The way that methods are combined depends on two factors:

- The designated type of method combination.

  Each generic function is associated with a type of method combination. By default this is **:daemon**. You can use the **:method-combination** option to **defflavor** or **defgeneric** to specify a different mechanism.

- The designated method order.

  By default this is **:most-specific-first**. In most types of method combination you can specify **:most-specific-last** to the **:method-combination** option.

The type of method combination and the method order are sufficient to choose which methods from the set of available methods should be run, and in what order they are to be run.

For example, the **:and** method combination type chooses all available primary methods and combines them inside an **:and** special form. Any **:before** or **:after** methods cause an error.

The order in which the methods are executed is important. This is determined by the ordering of flavor components and the method order used. When flavors are built from components, the flavor system computes a total ordering of components. For example, three flavors are defined as follows:

```
(defflavor flav-1 () (flav-2 flav-3))
(defflavor flav-2 () ())
(defflavor flav-3 () ())
```

The ordering of flavor components for the flavor **flav-1** is:

```
(flav-1 flav-2 flav-3)
```

flav-1 is said to be the most specific flavor in the ordering. Thus if **:most-specific-first** method order is used with the **:and** method combination type, and all three flavors define a method for the same generic function, the combined method resembles:

```
(and (method-for-flav-1)
     (method-for-flav-2)
     (method-for-flav-3))
```

If **:most-specific-last** method order is used with the **:and** method combination type, the combined method resembles:

```
(and (method-for-flav-3)
     (method-for-flav-2)
     (method-for-flav-1))
```

For information on how the flavor ordering is computed: See the section "Ordering Flavor Components".

For information on other built-in types of method combination, and how to define new types of method combination: See the section "Method Combination".

Inheritance of Instance Variables

When you define a flavor that is built on other flavors, the new flavor inherits instance variables from each of its component flavors. When two or more of the components have an instance variable with the same name, the new flavor inherits exactly one instance variable with that name. All components of the new flavor share this variable. The default initial value for an instance variable comes from the first flavor in the ordering of flavor components that specifies a value. For example:

```
;;; the basic flavor in the family of printers
;;; all printers must have a name and baud-rate
(defflavor basic-printer (name baud-rate)
           ()
  :initable-instance-variables
  (:required-init-keywords name baud-rate))

;;; line-printers in this shop run at 1200 baud
(defflavor line-printer ()
           (basic-printer)           ;built on basic-printer
  (:default-init-plist :baud-rate 1200))

;;; smart printers know their own status and length of queue
;;; in this shop they run at 2400 baud
(defflavor smart-printer (current-status queue)
           (basic-printer)            ;built on basic-printer
  :initable-instance-variables
  (:default-init-plist :baud-rate 2400
                       :queue 0))
```

In this flavor family, all types of printers have instance variables for **name** and **baud-rate**. The flavors **smart-printer** and **line-printer** inherit the instance variables **name** and **baud-rate** from their component flavor **basic-printer**.

The flavor **basic-printer** specifies that the **name** and **baud-rate** instance variables are required to be initialized, when making an instance of **basic-printer** or either of the two flavors that are built on it.

The flavor **line-printer** has only one difference from **basic-printer**; it specifies a default initial value for the **baud-rate** instance variable.

The flavor **smart-printer** supplies two additional instance variables, **current-status** and **queue**. It also specifies a default initial value for **queue** and for **baud-rate**.

Inheritance of defflavor Options

A flavor built from components inherits characteristics from those components, including many of the **defflavor** options. This section describes which **defflavor** options are inherited, and which are not.

:abstract-flavor Not inherited.

:area-keyword Inherited. If more than one component specifies this option, the **:area-keyword** chosen comes from the most specific flavor in the ordering.

:component-order Inherited. Any ordering restrictions noted in a flavor's **:component-order** option are considered as ordering restrictions by all flavors that are built on it.

:conc-name Not inherited.

:constructor Not inherited.

:default-handler Inherited.

:default-init-plist Inherited. A flavor's **:default-init-plist** is the union of its own, and those of its components. If any elements of the **:default-init-plist** are in conflict, the conflict is resolved by choosing the element from the most specific flavor that provides it.

:documentation Not inherited.

:functions Inherited.

:gettable-instance-variables
Inherited. Any instance variables specified as gettable by a flavor component are also gettable by the flavor that is built on that component.

:init-keywords Inherited. A flavor's list of **:init-keywords** is the union of the **:init-keywords** of all of its components.

:initable-instance-variables
Inherited. Any instance variables specified as initable by a flavor component are also initable by the flavor that is built on that component.

:locatable-instance-variables
Inherited. Any instance variables specified as locatable by a flavor component are also locatable by the flavor that is built on that component.

:method-combination
Inherited. If **:method-combination** is specified more than once for the same generic function by flavor components, they must agree on the type of method combination and the parameters.

:method-order Inherited. A flavor's method order is the resulting of appending the method-orders of all its components, in the order of flavor components.

:mixture Not inherited.

:no-vanilla-flavor Inherited.

:ordered-instance-variables
Inherited. If this is specified by more than one component, the order of the instance variables must agree. For example, if one flavor specifies (**:ordered-instance-variables a b c**), another component flavor could legally specify (**:ordered-instance-variables a b c d e**). However it would be illegal for a component flavor to specify (**:ordered-instance-variables d e**).

:readable-instance-variables
Inherited. Any instance variables specified as readable by a flavor component are also readable by the flavor that is built on that component.

:required-flavors Inherited. A flavor's list of **:required-flavors** is the union of the **:required-flavors** of all of its components.

:required-init-keywords
Inherited. A flavor's list of **:required-init-keywords** is the union of the **:required-init-keywords** of all of its components.

:required-instance-variables
Inherited. A flavor's list of **:required-instance-variables** is the union of the **:required-instance-variables** of all of its components.

:required-methods Inherited. A flavor's list of **:required-methods** is the union of the **:required-methods** of all of its components.

:settable-instance-variables
Inherited. Any instance variables specified as settable by a flavor component are also settable by the flavor that is built on that component.

:special-instance-variables
Inherited. A flavor's list of **:special-instance-variables** is the union of the **:special-instance-variables** of all of its components.

:special-instance-variable-binding-methods
Inherited. A flavor's list of **:special-instance-variable-binding-methods** is the union of the **:special-instance-variable-binding-methods** of all of its components.

:writable-instance-variables
Inherited. Any instance variables specified as writable by a flavor component are also writable by the flavor that is built on that component.

The Vanilla Flavor

By default, every flavor includes the flavor called **flavor:vanilla**. **flavor:vanilla** has no instance variables, but it provides several basic useful methods, some of which are used by the Flavor tools. See the section "Generic Functions and Messages Supported by **flavor:vanilla**".

You can specify not to include **flavor:vanilla** by providing the **:no-vanilla-flavor** option to **defflavor**. This would be unusual.

Ordering Flavor Components

This section describes how Flavors determines the ordering of flavor components, when a new flavor is built on component flavors. You can view the order using Show Flavor Components: See the section "Show Flavor Commands".

flavor:get-all-flavor-components returns the list of ordered components: See the function **flavor:get-all-flavor-components**.

The **defflavor** forms of a flavor and its components set local constraints on the ordering of flavor components. When a flavor is built from components, all of the local constraints of the flavor and its components are taken into account, and an ordering is computed that satisfies all of these constraints. In other words, the **defflavor** forms specify partial orderings, which must be merged into one total ordering.

Three rules govern the ordering of flavor components:

1. A flavor always precedes its own components.

2. The local ordering of flavor components is preserved; a component precedes all components that appear to its right in the list of flavor components.

3. Duplicate flavors are eliminated from the ordering; if a flavor appears more than once, it is placed as close to the beginning of the ordering as possible, while still obeying the other rules.

In many cases, these three rules are enough to define a unique ordering of flavor components. In other cases, the rules result in several possible orderings, and Flavors applies another guideline:

A tree of flavors is constructed from the list of flavor components. The ordering is determined by walking through the tree of flavors in depth-first order, and adding each node (flavor) to the ordering if it fits the constraints of the three rules.

The first node (the root of the tree) is the first flavor in the ordering, provided that it does not transgress any of the rules. The Flavor system continues to the next node, traversing the tree in depth-first order. If the node can be placed next in the ordering without transgressing one of the three rules, it is added it to the ordering. If adding it would transgress a rule, that node is skipped. If the end of the tree is reached and some flavors have not yet been placed in the ordered list, the Flavor system walks through the tree again, applies the three rules to the re-

maining nodes and adds them to the ordering. In complicated programs, it is conceivable that we could walk through the tree several times.

The following examples illustrate how we could predict how new Flavors would choose an ordering of components of **pie**, which is defined as follows:

Simple Example of Ordering Flavor Components

In this example, the three rules define exactly one ordering for **new-flavor**:

```
(defflavor new-flavor () (apple tomato))
(defflavor apple () (fruit))
(defflavor tomato () (fruit vegetable))
(defflavor fruit () ())
(defflavor vegetable () ())
```

To illustrate how the rules apply to this example:

1. A flavor always precedes its own components.

   - **new-flavor** precedes **apple** and **tomato**.
   - **apple** precedes **fruit**.
   - **tomato** precedes **fruit** and **vegetable**.

2. The local ordering of flavor components is preserved.

   - **apple** precedes **tomato**.
   - **fruit** precedes **vegetable**.

The **flavor:vanilla** flavor is included in all flavors unless it is explicitly excluded (by the **:no-vanilla-flavor** option to **defflavor**). The first rule constrains **flavor:vanilla** to appear last in every ordering of flavors, if it is present.

The resulting ordering of flavors is:

```
(new-flavor apple tomato fruit vegetable flavor:vanilla)
```

Using The Tree to Order Components

In this example, the three rules do not define a single ordering for **pie**. The flavors are defined as follows:

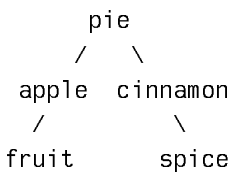
```
(defflavor pie () (apple cinnamon))
(defflavor apple () (fruit))
(defflavor cinnamon () (spice))
(defflavor fruit () ())
(defflavor spice () ())
```

Three orderings are possible, under the rules:

```
(pie apple fruit cinnamon spice flavor:vanilla)
(pie apple cinnamon fruit spice flavor:vanilla)
(pie apple cinnamon spice fruit flavor:vanilla)
```

A well-conceived program should not depend on any one of those orderings, but should work equally well under any of them. For example, if your program depends on **spice** preceding **fruit** in the ordering for **pie**, you should make that constraint explicit, by including those two flavors in the list of components in the **defflavor** form for **pie**.

The following text describes how to predict the ordering that the Flavors system would choose, when several orderings are possible. We construct a tree of flavor components:



We order the components by walking through the tree in a depth-first order, adding the components to the order, always checking that no rules are transgressed. The resulting ordering is:

```
(pie apple fruit cinnamon spice flavor:vanilla)
```

Example of an Inconsistent Set of Flavor Definitions

It is possible to write a set of flavor definitions that cannot be ordered using the rules. For example:

```
(defflavor new-flavor () (fruit apple))
(defflavor apple () (fruit))
```

To illustrate how the rules apply to this example:

1. A flavor always precedes its own components.

   - **new-flavor** precedes **fruit** and **apple**.
   - **apple** precedes **fruit**.

2. The local ordering of flavor components is preserved.

   - **fruit** precedes **apple**

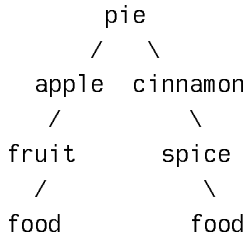
Two of the rules contradict each other: **apple** precedes **fruit**, and **fruit** precedes **apple**. When this situation occurs, Flavors signals an error when it tries to compute the ordering (usually the first time you call **compile-flavor-methods** or **make-instance**). The error message includes a detailed description of the minimal set of conflicting constraints that cause the inconsistency. At that point you can redefine one or more of the flavors to resolve the problem, either by changing the order of flavor components so there is no conflict, or by using the **:component-order** option to **defflavor** to relax the ordering constraints.

Skipping a Flavor When Traversing the Tree

This example illustrates the case when a flavor cannot be added to the ordering the first time it appears in the tree:

```
(defflavor pie () (apple cinnamon))
(defflavor apple () (fruit))
(defflavor cinnamon () (spice))
(defflavor fruit () (food))
(defflavor spice () (food))
(defflavor food () ())
```

We construct a tree of flavors:



The list begins with **pie**. We continue, adding **apple** and **fruit**. So far, the (incomplete) ordered list is:

```
(pie apple fruit
```

The next node is **food**, but we cannot place it next in the ordering because doing so would transgress the rule that a flavor always precedes its own components. If placed next, **food** would precede **spice**, and we know that **spice** must precede **food**. Thus we skip **food** and continue through the tree. Because **food** appears later in the tree, we pick it up then. The ordering is now complete:

```
(pie apple fruit cinnamon spice food flavor:vanilla)
```

In more complex cases, we might finish walking through the tree without picking up all the flavors. In that case, we would walk through the tree again, picking up the remaining flavors as long as doing so would not transgress the rules. It is conceivable that we might need to walk through the tree several times, if many related flavors are mixed.

Final Note

Note the following example:

```
(defflavor pie () (apple cinnamon))
(defflavor pastry () (cinnamon apple))
```

The ordering for **pie** is:

```
(pie apple cinnamon flavor:vanilla)
```

The ordering for **pastry** is:

```
(pie cinnamon apple flavor:vanilla)
```

There is no problem with the fact that **apple** precedes **cinnamon** in the ordering of the components of **pie**, but not in the ordering for **pastry**. However, you cannot build a new flavor that has both **pie** and **pastry** as components.

Flavor Families

The following organization conventions are recommended for programs that use flavors:

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have that base flavor as one of their components. Typically the base flavor includes things relevant to the whole family, such as instance variables, **:required-methods** and **:required-instance-variables** declarations, default methods for certain generic functions, **:method-combination** declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most cannot be instantiated and merely serve as a foundation on which to build other flavors. The base flavor for the *foo* family is often named **basic-***foo*.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin. You construct usable flavors by choosing the mixins for the desired characteristics and combining them with the base flavor. A mixin flavor that provides the *mumble* feature is often named *mumble*-**mixin**.

An instantiatable flavor combines several mixins with a base flavor to produce desired behavior. It is a complete description that can be used with **make-instance**.

By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor components controls the order in which methods are combined. Order dependencies should be documented as part of the conventions of the appropriate family of flavors.

You can follow these conventions by naming the basic flavors and mixin flavors as suggested, and using **defflavor** to define usable flavors composed of a handful of mixins and the basic flavor. **defflavor** also offers several options that can help organize flavor families:

- **:abstract-flavor** specifies that a flavor cannot be instantiated.

- **:component-order** states explicitly the order restrictions you want imposed on component flavors. When many mixins are combined together, the normal ordering constraints can cause conflicts that make it impossible to compute a total ordering of flavor components. If you know that several of the components should have ordering constraints but that others need not, you can relax the ordering restrictions by using **:component-order**.

- **:documentation** lets you document the organization of the flavor family.

- **:mixture** lets you write a framework with rules that define a flavor family. This is a more structured way to organize a flavor family.

- **:required-methods**, **:required-instance-variables**, and **:required-flavors** let you explicitly state the requirements of flavors that are to be instantiated.

Example of Programming with Flavors: Life

This section contains an annotated program that illustrates several principles of using Flavors. (This code is in `sys:examples;flavor-life.lisp`.) Because Flavors is intended to help organize large, complex programs, it is difficult to exercise all (or even most) of the features of Flavors in a small demonstration program. However, this program does illustrate the following aspects of Flavors:

- Using **defflavor**, **defgeneric**, **defmethod**, and **make-instance**
- Writing a method for **make-instance**
- Constructing a flavor from component flavors
- Using **:and** type of method combination
- Integrating flavors with other Lisp data structures (arrays)
- Using message-passing (**send**) when necessary to use old interfaces

We will develop a program that plays the game of "Life." Life simulates a community of beings called cells. The rules of Life indicate whether a given cell will live or die in the next generation, depending on its environment. If the cell is too crowded by its neighbors, or too isolated from other cells, it dies. Otherwise the environment is deemed acceptable, and the cell lives. Specifically:

- If an empty cell has exactly 3 live neighbors, a cell is born there.
- If an empty cell has any other number of live neighbors, no cell is born.
- If a live cell has 2 or 3 live neighbors, it stays alive.
- If a live cell has any other number of live neighbors, it dies.

Some implementations of Life have a finite but unbounded domain, by treating a cell that lives on the edge of the domain as a neighbor of the corresponding cell on the other edge of the domain. For the purpose of this example we have a simpler version of Life, characterized by the following statements:

- Cells live in a two-dimensional array.
- A cell typically has eight neighbors, those adjacent to it.
- Cells on the border of the array have less than eight neighbors.

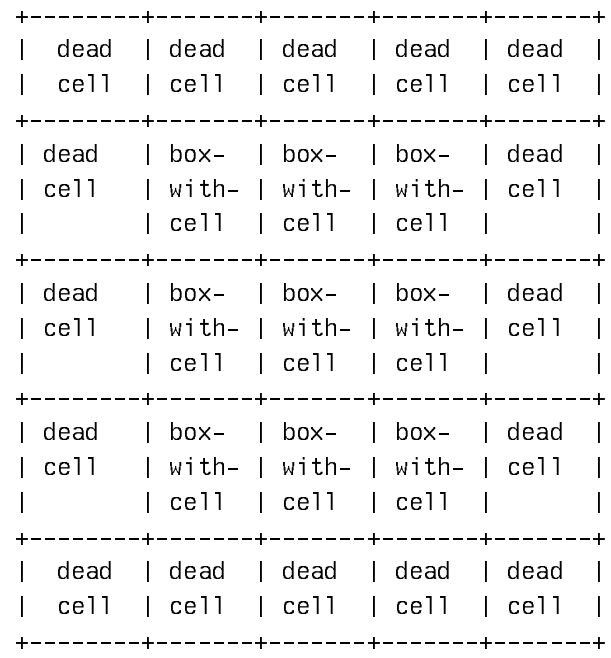
Note that this example program is intended to illustrate Flavors, and is not attempting to run Life in the fastest possible way.

Organizing the Program

Three flavors are used to represent the Life cells: **cell**, **box**, and **box-with-cell**. The **cell** flavor is flexible enough that you could use it in another program that uses a different type of display. The **box** flavor is used to display the Life gameboard on the screen. **box-with-cell** combines those two flavors.

The following illustration shows the Life gameboard implemented by this program. In this illustration, the variable ***number-boxes-on-axis*** is 3; that is, the Life game is played on a 3 by 3 array. (In the program below, ***number-boxes-on-axis*** is much larger than 3.) Each instance of **box-with-cell** is a unit of the Life game, and displayed in each generation. To make it easier to compute live neighbors of each **box-with-cell**, we have an invisible border composed of instances of **cell** (all dead). This way, each **box-with-cell** has eight neighbors. The border cells are not displayed. Only the inner part of the gameboard is displayed for the user.

Life Gameboard



Defining Variables Used in the Program

The program begins by defining and initializing special variables:

```
;;; This array is used for storing cells and box-with-cells.
(defvar *game-board*   nil
  "array for the Life game.")
(defvar *number-boxes-on-axis*  30
  "number of boxes on each axis of gameboard")

;;; This window is used for displaying the *game-board*.
```

```
(defvar *game-window*  (tv:make-window 'tv:window
                                       :blinker-p nil
                                       :label "Game of Life")
  "Window for display.")

;;; The following numbers make a nice display:
(defvar *x-top-left-corner* 250
   "x coordinate of top left corner of display")
(defvar *y-top-left-corner* 100
  "y coordinate of top left corner of display")
(defvar *side-length* 15
  "length of each box for display")
(defvar *board-length*  (* *side-length* *number-boxes-on-axis*)
  "length of gameboard for display")
(defvar *cell-radius* 6
  "radius of circle to draw live cells")
```

Defining the Flavors

```
;;; A cell is a functional cell of the Life game.  It stores
;;; its status and next-status (:alive or :dead), and a list of
;;; its neighbors.   The x and y instance variables give the
;;; coordinates of this cell in the array *game-board*.

(defflavor cell (x y status next-status neighbors) ()
  (:initable-instance-variables x y status))

;;; A box is intended only to be displayed on the screen.
;;; box-x and box-y are the coordinates of its top-left corner.
;;; side-length is the length of one side of the box.

(defflavor box (box-x box-y box-x-center box-y-center side-length)
           ()
  (:initable-instance-variables box-x box-y side-length))

;;; A box-with-cell is a box that contains a functional Life cell.
;;; The cell component provides the necessary methods for playing Life.
;;; The box component makes it easy to display the game-board.
;;; box-with-cell has default initial values for our game.

(defflavor box-with-cell ()                    ;no instance vars
           (box cell)                          ;two components
  (:default-init-plist :side-length *side-length*
                       :status (if (evenp (random 2)) ':alive ':dead))
  (:required-methods aliveness count-live-neighbors
                     get-next-status change-status
                     draw-outline))
```

Defining the Generic Functions

Now we define the generic functions. The argument *cell-unit* implies that the generic function can be used on instances of **cell** or any flavor built on **cell**, such as **box-with-cell**. Similarly, the argument *box-unit* implies that the generic function **draw-outline** can be used on instances of **box** or **box-with-cell**. We use the **defgeneric** forms as the appropriate place to document the contract of the generic function. (One additional generic function, **change-status**, is defined later on in this section.)

```
(defgeneric aliveness (cell-unit)
  "Returns 1 if the cell-unit is currently alive, 0 otherwise.")

(defgeneric find-neighbors (cell-unit)
   "Calculates and stores the 8 neighbors of a cell-unit.")

(defgeneric count-live-neighbors (cell-unit)
   "Returns the number of live neighbors of a cell-unit.")

(defgeneric get-next-status (cell-unit)
  "Applies rules of the Life game, using count-live-neighbors.
   If overcrowded or too isolated, this cell-unit dies.
   If the environment is fine, this cell-unit lives.
   The result is remembered by the cell-unit."

(defgeneric draw-outline (box-unit)
  "Draws the outline of the given box-unit.")

(defgeneric draw-contents (box-with-cell)
  "Draws the cell contained in a box-with-cell.
   Live cells appear as filled-in circles; dead cells are invisible.")
```

Defining the Methods

We now define the methods that implement the generic functions on instances of **cell** and **box**. These methods are inherited by **box-with-cell**. Notice that the methods for **draw-outline** and **draw-contents** use message-passing to invoke methods associated with the window system. There is no difficulty in writing programs that use both generic functions and message-passing. (The methods for **change-status** are defined later on in this section).

```
;;; aliveness returns 1 if cell is alive, 0 otherwise

(defmethod (aliveness cell) ()
  (if (eq status ':alive) 1 0))

;;; the neighbors of a cell are its 8 adjacent cells
;;; because there is a border of dead cells, in this implementation
;;; every box-with-cell has 8 neighbors.
```

```
(defmethod (find-neighbors cell) ()
  (setq neighbors (list
                    (aref *game-board* x (1- y))
                    (aref *game-board* x (1+ y))
                    (aref *game-board* (1- x) y)
                    (aref *game-board* (1+ x) y)
                    (aref *game-board* (1- x) (1- y))
                    (aref *game-board* (1- x) (1+ y))
                    (aref *game-board* (1+ x) (1- y))
                    (aref *game-board* (1+ x) (1+ y)))))

(defmethod (count-live-neighbors cell) ()
  (reduce #'+ (map 'list #'aliveness neighbors)))

(defmethod (get-next-status cell) ()
  (let ((number-live-neighbors (count-live-neighbors self)))
    (setq next-status
          (cond ((eq ':dead status)             ;empty cell
                 (if (= number-live-neighbors 3)
                     ':alive
                     ':dead))
                (t                              ;live cell
                 (if (or (= number-live-neighbors 2)
                         (= number-live-neighbors 3))
                     ':alive
                     ':dead))))))

;;; draw-outline uses message-passing (send function)
;;; to invoke the :draw-line method

(defmethod (draw-outline box) ()
  (send *game-window* :draw-line
        box-x box-y
        (+ box-x side-length) box-y)
  (send *game-window* :draw-line
        box-x box-y
        box-x (+ box-y side-length))
  (send *game-window* :draw-line
        (+ box-x side-length) box-y
        (+ box-x side-length) (+ box-y side-length))
  (send *game-window* :draw-line
        box-x (+ box-y side-length)
        (+ box-x side-length) (+ box-y side-length)))

(defmethod (draw-contents box-with-cell) ()
  (if (= (aliveness self) 1)
      ;; draw a circle to represent an alive cell
```

```
      (send *standard-output* :draw-filled-in-circle
            box-x-center box-y-center *cell-radius*)
      ;; erase a circle if the cell is dead
      (send *standard-output* :draw-filled-in-circle
            box-x-center box-y-center *cell-radius* tv:alu-andca)))
```

Using :and Method Combination

change-status illustrates the use of **:and** method combination. The generic function **change-status** has two methods written for it: one method implements **change-status** on instances of **cell**; the other method implements **change-status** on instances of **box-with-cell**. Because the **:and** type of method combination is specified in the **defgeneric** form, the handler for instances of **box-with-cell** is a combined method. The combined method first executes the method for **cell** (since **:most-specific-last** order is used). If that method returns non-**nil**, the method for **box-with-cell** is executed. This enables us to redisplay the contents of the cell only if necessary; that is, only if the status of the cell has changed.

```
(defgeneric change-status (cell-unit)
   "When applied to a cell, updates it to next-status.
    When applied to a box-with-cell, checks to see if
    the status changed.   If so, redisplays the contents."
   (:method-combination :and :most-specific-last))

;;; The following method is inherited by box-with-cell,
;;; combined with the :and type of method combination.
;;; The return value is important because it determines
;;; whether or not the box-with-cell needs to be redisplayed:

(defmethod (change-status cell) ()
  (if (eq status next-status)
      nil                              ;returns nil if no change
      (setq status next-status)))      ;returns non-nil if status changed

(defmethod (change-status box-with-cell) ()
  (draw-contents self))

;;;Note that the combined method for change-status of a box-with-cell
;;;looks like this:
;;;      (and (method for cell)
;;;           (method for box-with-cell))
```

Writing a Method for make-instance

We now write a method for **make-instance** of **box-with-cell**. This method is run every time a new instance of **box-with-cell** is made. It does some further initialization of the new instance, depending on the fact that the instance variables **box-x** and **box-y** are initialized.

```
(defmethod (make-instance box-with-cell) (&rest ignore)
  (setq box-x-center (round (+ box-x (* .5 *side-length*))))
  (setq box-y-center (round (+ box-y (* .5 *side-length*)))))
```

Using Flavors in Conjunction with an Array

Now that we have defined the flavors, generic functions, and methods for Life, we need only put the pieces together to complete the program:

```
;;; play-life-game is the top-level function that plays the Life game.

(defun play-life-game (&optional (generations 3))
  (set-up-game-board)                       ;initialize gameboard
  (iterate-game-board #'find-neighbors)
  (iterate-game-board #'draw-outline)       ;display gameboard grid
  (iterate-game-board #'draw-contents)      ;display initial set-up
  (loop for i from 1 to generations
        do
    (iterate-game-board #'get-next-status)  ;compute next-status
    (iterate-game-board #'change-status)))  ;update status & display

;;; *game-board* is a 2-dimensional array:
;;; outer border contains dead cells (easier to compute live neighbors)
;;; outer border is not displayed
;;; inner part contains box-with-cells (dead or alive by random)

(defun set-up-game-board ()
  (setq *game-board* (make-array (list (+ *number-boxes-on-axis* 2)
                                       (+ *number-boxes-on-axis* 2))))

  ;; initialize the border with dead cells
  (loop for x-pos from 0 to (1+ *number-boxes-on-axis*)
        do
      (setf (aref *game-board* x-pos 0)
            (make-instance 'cell :status ':dead))
      (setf (aref *game-board* x-pos (1+ *number-boxes-on-axis*))
            (make-instance 'cell :status ':dead)))
  (loop for y-pos from 0 to (1+ *number-boxes-on-axis*)
        do
      (setf (aref *game-board* 0 y-pos)
            (make-instance 'cell :status ':dead))
      (setf (aref *game-board* (1+ *number-boxes-on-axis*) y-pos)
            (make-instance 'cell :status ':dead)))

  ;; now initialize the inner part of the array with box-with-cells
  (loop for x-pos from 1 to *number-boxes-on-axis*      ;inner part
          for x-offset from 0 by *side-length*
        do
```

```
    (loop for y-pos from 1 to *number-boxes-on-axis*
          for y-offset from 0 by *side-length*
          do
      ;; fill up with box-with-cells
      (setf (aref *game-board* x-pos y-pos)
            (make-instance 'box-with-cell
                           :x x-pos
                           :y y-pos
                           :box-x (+ *x-top-left-corner* x-offset)
                           :box-y (+ *y-top-left-corner* y-offset))))))

;;; iterate-game-board accesses the inner part of *game-board* and
;;; applies operation to each box-with-cell.   Any operation that can be
;;; used on a single cell or box-with-cell can be used.   For example:
;;;    draw-outline, draw-contents, get-next-status, change-status

(defun iterate-game-board (operation)
  (loop for y from 1 to *number-boxes-on-axis*
        do                                ;inner part of *game-board*
    (loop for x from 1 to *number-boxes-on-axis*
          do
      (funcall operation (aref *game-board* x y)))))
```

Compiling Flavor Methods

The last line in the file that contains this program is:

```
(compile-flavor-methods cell box box-with-cell)
```

This causes the combined methods for the three flavors to be compiled at compile-time, and the data structures to be built at load-time, rather than both happening at run-time. This speeds up the program considerably the first time it is run.

Flavors Tools

Flavors stores a lot of information about defined flavors, generic functions, and methods in its internal framework. Flavors tools are designed to give you access to that information in a form useful for program development and debugging.

The tools are divided into three groups:

- Show Flavor Commands

  The Show Flavor commands provide a variety of powerful tools for developing, debugging, and understanding Flavors-based programs. You can use these commands in the command processor, the editor, and the Flavor Examiner. In a dynamic window, you can also specify them using the mouse. Click Right on a displayed instance, flavor name, generic function name, or method name for a menu of Show Flavor commands.

- Zmacs Commands for Flavors

  These Zmacs commands help you deal with issues that come up when you are editing definitions of flavors, methods, and generic functions. These commands are available only in the editor.

- The Flavor Examiner

  This is an environment for using the Show Flavor commands. Use SELECT X for the Flavor Examiner.

Summary of Show Flavor Commands

Genera provides the following tools for analyzing Flavors-based programs:

Show Flavor Components *flavor keywords...*
: Answers: What is the order of flavor components, and why did the system pick that order?

Show Flavor Dependents *flavor keywords...*
: Answers: What flavors inherit from this one?

Show Flavor Differences *flavor-1 flavor-2 keywords...*
: Answers: What are the differences between two flavors?

Show Flavor Functions *flavor keywords*
: Answers: What internal flavor functions are defined for this flavor?

Show Flavor Handler *operation flavor keywords...*
: Answers: When an operation (generic function or message) is applied to an instance of a given flavor, what methods implement the operation? What method combination type is used? What is the order of methods in the handler?

Show Flavor Initializations *flavor keywords...*
: Answers: How are new instances of this flavor initialized?

Show Flavor Instance Variables *flavor keywords...*
: Answers: What state is maintained by instances of this flavor?

Show Flavor Methods *flavor keywords...*
: Answers: What methods are defined for this flavor, or inherited from its component flavors?

Show Flavor Operations *flavor keywords...*
: Answers: What operations (generic functions and messages) are supported by instances of this flavor?

Show Generic Function *operation keywords...*
: Answers: What are the general characteristics of this generic function or message? What flavors provide a method for it? What methods are implemented for it?

These commands accept keywords that modify their behavior. Keyword options are available to request information in brief form, in detailed form, sorted by flavor, and so on. See the section "Keyword Options for Show Flavor Commands".

For details of each of these commands, and examples, see the section "Show Flavor Commands".

Entering Input for Show Flavor Commands

The Show Flavor commands are integrated with the Dynamic Lisp Listener environment.

To give a Show Flavor command you can:

- Type the name of the command in the command processor. Completion is offered for the command names.

- Highlight a flavor, instance, generic function, or method with the mouse. Now when you click Right, a menu of Show Flavor commands appears.

To enter arguments to the commands you can:

- Type the desired argument. Completion is supported. You can include or omit any package prefixes.

- Point with the mouse to the desired object on the screen (such as a flavor, generic function, or method) and click Left.

Note that input you have typed (such as a flavor name) is not mouse-sensitive. The output of a previous Show Flavor command is mouse-sensitive.

Output of Show Flavor Commands

The output of Show Flavor commands is mouse-sensitive. Sometimes the output is abbreviated. For example, a method might be represented by the name of the flavor that provides it. Even when the representation of the method is abbreviated, the method is mouse-sensitive.

You can use the output by clicking Left on the representation of an object to enter it as input to another Show Flavor command, or clicking Right to get a menu of commands. This way you can use the output of previous commands to continue to explore the universe of defined flavors.

Keyword Options for Show Flavor Commands

This section gives a general description of the keyword options accepted by most Show Flavor commands. Some of the keywords have a different meaning for each command. The documentation on the individual commands states which keywords are accepted, and describes the meaning of the keywords for that command (if they differ from the meaning described here).

Altering the Output Format

:sort — {alphabetical, flavor} Indicates how to sort the display. Alphabetical means the output is sorted alphabetically, ignoring package prefixes. For example, the output of Show Flavor Methods is sorted alphabetically by generic function name. Alphabetical is the default. If flavor is specified, the output is sorted according to the order of flavor components.

Restricting the Output

:brief — {yes, no} If yes, requests a brief answer to the question asked. The exact meaning of :brief varies for each command.

:locally — {yes, no} If yes, specifies that inherited characteristics are not to be shown.

:match — {*string*} Requests only those things (flavors, methods, generics, or whatever was requested) that match this substring. If *string* is omitted, requests all of them. :match has a different meaning for each command. This option lets you pare down the output.

:More Processing — {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

Requesting Additional Output

:detailed — {yes, no} If yes, requests a detailed answer to the question asked. The exact meaning of :detailed varies for each command.

:functions — {*string*} Requests internal functions to flavors that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no internal functions are requested.

:initializations — {*string*} Requests initializations that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no initializations are requested.

:instance variables — {*string*} Requests instance variables that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no instance variables are requested.

:methods {*string*} Requests methods for generics that match this substring. If *string* is omitted, requests all of them. If the keyword itself is not supplied, no methods are requested.

Redirecting the Output

:Output Destination {Buffer, File, Kill Ring, None, Printer, Stream, Window} Enables you to direct your output. The default is the stream ***standard-output***. Note that redirecting output to a printer can be particularly useful.

Show Flavor Commands

The following commands show attributes of a flavor, generic function, method, or handler. Only those keywords that are specific to each command (or have a different meaning for each command) are explained in the command descriptions. For an explanation of any keywords not covered in the command descriptions, see the section "Keyword Options for Show Flavor Commands".

Show Flavor Components Command

Show Flavor Components *flavor keywords*

Shows the order of the components of this flavor.

keywords :Brief, :Detailed, :Duplicates, :Functions, :Initializations, :Instance Variables, :Match, :Methods, and :Output Destination. See the section "Keyword Options for Show Flavor Commands".

:Duplicates {Yes, No} Indicates whether or not to display duplicate occurrences of flavors. The default is No.

:Brief {Yes, No} Yes indicates that the output should not be indented to show the structure. The default is No.

The flavor components are ordered from top to bottom. The top flavor is the *most specific flavor* in the ordering. The indentation graphically represents which flavors are components of which other flavors. In the example below, **tv:minimum-window** has six direct components: **tv:essential-expose**, **tv:essential-activate**, **tv:essential-set-edges**, **tv:essential-mouse**, **tv:essential-window**, and **flavor:vanilla**.

When you use the :duplicates keyword and show the components of complex flavors, you notice special symbols in the display. For example:

```
Command: Show Flavor Components TV:MINIMUM-WINDOW :Duplicates
    --> TV:MINIMUM-WINDOW
          TV:ESSENTIAL-EXPOSE
            [TV:ESSENTIAL-WINDOW] ↓
          TV:ESSENTIAL-ACTIVATE
            [TV:ESSENTIAL-WINDOW] ↓
          TV:ESSENTIAL-SET-EDGES
            [TV:ESSENTIAL-WINDOW] ↓
          TV:ESSENTIAL-MOUSE
          TV:ESSENTIAL-WINDOW
            TV:SHEET
              SI:OUTPUT-STREAM
                SI:STREAM
          FLAVOR:VANILLA
```

Bracketed flavors are duplicates that are included by the parent flavor here, but are not ordered in this position because of some ordering constraint. They appear in another place in the display without brackets, in their correct order. All bracketed components have an arrow beside them. A down-arrow indicates that this component's position in the ordering is later in the display. An up-arrow indicates that this component's position in the ordering is earlier in the display; these occurrences are infrequent.

For example, the flavor **tv:essential-window** is a component of four other components: **tv:essential-expose**, **tv:essential-activate**, **tv:essential-set-edges**, and **tv:minimum-window** itself. Its correct position in the ordering is directly after **tv:essential-mouse**, where it appears without brackets.

You can read the order of flavor components by reading all unbracketed flavors from top to bottom, ignoring punctuation. If :Duplicates is No, this is all that is displayed.

For information on how the order is determined, see the section "Ordering Flavor Components".

Show Flavor Dependents Command

Show Flavor Dependents *flavor keywords*

Shows the names of flavors that are dependent on this flavor.

keywords — :Brief, :Detailed, :Duplicates, :Functions, :Initializations, :Instance Variables, :Levels, :Match, :Methods, :Output Destination. See the section "Keyword Options for Show Flavor Commands".

:Brief — {Yes, No} Yes indicates that the output should not be indented to show the structure. The default is No.

:Duplicates {Yes, No} Indicates whether or not to display duplicate occurrences of flavors. The default is No.

:Levels {All, *integer*} Specifies how many levels of indirect dependency to display. The default is all, which shows all levels. For some flavors the output can be voluminous, and it is helpful to use :Levels to pare it down.

A dependent flavor is a flavor that uses this flavor as a component (directly or indirectly). This is useful in program development or debugging, to answer the question "What flavors will be affected if I change the definition of this flavor?" For example:

```
Command: Show Flavor Dependents TV:SCROLL-WINDOW-WITH-DYNAMIC-TYPEOUT
     --> TV:SCROLL-WINDOW-WITH-DYNAMIC-TYPEOUT
           TV:BASIC-PEEK
             TV:PEEK-PANE
           TV:BASIC-TREE-SCROLL
             LMFS:AFSE-MIXIN
               LMFS:FSMAINT-AFSE-PANE
             LMFS:FSMAINT-HIERED-PANE
             TV:MOUSABLE-TREE-SCROLL-MIXIN
               TV:TREE-SCROLL-WINDOW
```

The output is indented to clarify which flavor is built on which component flavors. The structure of the output is the inverse of the output of Show Flavor Components. In this example, **tv:basic-peek** is a direct dependent of **tv:scroll-window-with-dynamic-typeout**, and **tv:peek-pane** is a direct dependent of **tv:basic-peek**.

Show Flavor Differences Command

Show Flavor Differences *flavor1 flavor2 keywords*

Shows the characteristics that two flavors have in common, and the characteristics in which they differ.

keywords :Match, :More Processing, :Output Destination. See the section "Keyword Options for Show Flavor Commands".

:Match {*string*} Displays only those generic functions or messages that match the given substring.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

This is most useful for two flavors that share many characteristics. Here is some sample output:

```
Command: Show Flavor Differences TV:ESSENTIAL-WINDOW TV:MINIMUM-WINDOW
     --> TV:ESSENTIAL-WINDOW and TV:MINIMUM-WINDOW have
         common components:
            flavors...
         TV:MINIMUM-WINDOW has other components:
            flavors...

         Differences in :ACTIVATE methods from TV:ESSENTIAL-WINDOW
                                           to TV:MINIMUM-WINDOW
            TV:SHEET before, primary,
            TV:ESSENTIAL-ACTIVATE after [added]

         Differences in handling of :BURY
            Flavor TV:ESSENTIAL-WINDOW does not handle :BURY
            Methods of TV:MINIMUM-WINDOW:
               TV:ESSENTIAL-EXPOSE wrapper, TV:ESSENTIAL-ACTIVATE
            more differences...
```

First, the common components are displayed. Second, the extra components of either (or both) flavors are displayed. Third, any differences in handling of generic functions are displayed.

In this example, **tv:minimum-window** has one method for **:activate** that **tv:essential-window** does not have: an **:after** method provided by flavor **tv:essential-activate**. The term `[added]` indicates that this method is defined for the second flavor but not for the first flavor. If the command had been given such that *flavor-1* was **tv:minimum-window** and *flavor-2* was **tv:essential-window**, the term would have been `[deleted]`. To interpret which flavor "adds" or "deletes" a method, look at the line that defines the perspective: `"Differences in :ACTIVATE methods from TV:ESSENTIAL-WINDOW to TV:MINIMUM-WINDOW"`.

When comparing two complex flavors, the output can be voluminous. You can use :Match to pare down the output so it answers a specific question. For example:

```
Command: Show Flavor Differences DYNAMIC-LISP-LISTENER SHEET :Match
         screen
    --> information about common and different components...
        Difference in handling of :FULL-SCREEN
          Method of DW::DYNAMIC-LISP-LISTENER: TV:ESSENTIAL-SET-EDGES
          Flavor TV:SHEET does not handle generic operation :FULL-SCREEN
        another difference...
        5 local functions found with no differences:
          TV:SCREEN-MANAGE-RESTORE-AREA
          TV:SCREEN-MANAGE-CLEAR-AREA
          TV:SCREEN-MANAGE-CLEAR-UNCOVERED-AREA
          TV:SCREEN-MANAGE-CLEAR-RECTANGLE
          TV:SCREEN-MANAGE-MAYBE-BLT-RECTANGLE

        15 differing local functions were found that did not
        contain the substring "screen".
```

Show Flavor Functions **Command**

Show Flavor Functions *flavor keywords*

Shows internal flavor functions for the given flavor.

keywords	:Locally, :Match, :Output Destination, and :Sort. See the section "Keyword Options for Show Flavor Commands".
:Locally	{Yes, No} If yes, inherited internal flavor functions are not shown. The default is no, which shows all internal flavor functions defined for this flavor or inherited by this flavor.
:Match	{*string*} Displays only those internal functions that match the given substring.

Internal flavor functions are defined by **defun-in-flavor**, **defmacro-in-flavor**, and **defsubst-in-flavor**. See the section "Defining Functions Internal to Flavors".

```
Command: Show Flavor Functions TV:WINDOW
    -->  TV:ADJUST-MARGINS
         SI:ANY-TYI-CHECK-EOF
         SI:ASSURE-INSIDE-INPUT-EDITOR
         others...
```

Show Flavor Handler Command

Show Flavor Handler *operation flavor keywords*

Provides information on the handler that performs *operation* (which can be a generic function or a message) on instances of *flavor*.

keywords :Code and :Output Destination. See the section "Keyword Options for Show Flavor Commands".

:Code {Yes, No, Detailed} Specifies whether the Lisp code of the handler should be displayed. The default is No. Yes displays a template that resembles the actual code of the handler. Detailed displays the actual code of the handler. This displays some internal functions and data structures of the Flavors system. For most purposes, Yes is more useful than detailed.

If the handler is a single method (not a combined method), its function spec is given:

```
Command: Show Flavor Handler CHANGE-STATUS CELL
     --> The handler for CHANGE-STATUS of an instance of CELL is
         the method (FLAVOR:METHOD CHANGE-STATUS CELL).
         The method-combination type is :AND :MOST-SPECIFIC-LAST.
```

If the handler is a combined method, the method combination type and order of methods are displayed. In the following example, the methods used in the combined method are represented by the names of the flavors that implement them. Even in this abbreviated format, the representation of the method is mouse-sensitive.

```
Command: Show Flavor Handler CHANGE-STATUS BOX-WITH-CELL
     --> The handler for CHANGE-STATUS of an instance of
         BOX-WITH-CELL is a combined method, with
         method-combination type :AND :MOST-SPECIFIC-LAST.
         The methods in the combined method, in order of
         execution, are: CELL, BOX-WITH-CELL
```

For combined methods, :Code Yes is useful. It requests a template that resembles the actual code of the handler:

```
Command: Show Flavor Handler CHANGE-STATUS BOX-WITH-CELL :Code yes
     --> The handler for CHANGE-STATUS of an instance of
         BOX-WITH-CELL is a combined method, with
         method-combination type :AND :MOST-SPECIFIC-LAST.
         (DEFUN (FLAVOR:COMBINED CHANGE-STATUS BOX-WITH-CELL)
                  (SELF SYS:SELF-MAPPING-TABLE FLAVOR::.GENERIC.
                   &REST FLAVOR::DAEMON-CALLER-ARGS.)
           (AND call (FLAVOR:METHOD CHANGE-STATUS CELL)
                call (FLAVOR:METHOD CHANGE-STATUS BOX-WITH-CELL)))
```

Show Flavor Initializations **Command**

Show Flavor Initializations *flavor keywords*

Shows the initialization keywords accepted by **make-instance** of this flavor, and any default initial values.

keywords :Detailed, :Locally, :Match, :Sort, and :Output Destination. See the section "Keyword Options for Show Flavor Commands".

:Detailed {Yes, No} The default is No, which requests the allowed initialization keywords that can be given to **make-instance** of this flavor, including init keywords and initable instance variables. If :Detailed is Yes, any additional instance variables are also shown; these are not initable instance variables. They are initialized by default values given in the **defflavor** form. Also, any initialization methods are shown. In other words, when :Detailed is No, you see the initializations from an external perspective (useful for making an instance). When :Detailed is Yes, you see the initializations from an internal perspective and gain information about how the flavor is constructed internally.

:Locally {Yes, No} If Yes, inherited initializations are not shown. The default is No, which requests all initializations defined for this flavor or inherited by this flavor.

:Match {*string*} Requests only those initializations matching the given substring.

For example:

```
Command: Show Flavor Initializations BOX-WITH-CELL :Detailed
    -->  Instances of BOX-WITH-CELL are created in the default area
         Another area can be specified with the keyword :AREA
         Initialization keywords that initialize
           instance variables:
             :BOX-X → BOX-X
             :BOX-Y → BOX-Y
             :SIDE-LENGTH → SIDE-LENGTH, default is *SIDE-LENGTH*
             :STATUS → STATUS, default is (IF (EVENP (RANDOM 2))
                                              ':ALIVE ':DEAD)
             :X → X
             :Y → Y
         Initialization method:
            MAKE-INSTANCE method: BOX-WITH-CELL
```

Show Flavor Instance Variables Command

Show Flavor Instance Variables *flavor keywords*

Shows the state maintained by instances of the given flavor.

keywords :Detailed, :Locally, :Match, :Output Destination and :Sort. See the section "Keyword Options for Show Flavor Commands".

:Detailed {Yes, No} If Yes, the attributes of the instance variables are shown, such as their accessibility or initializations. The default is No.

:Locally {Yes, No} If Yes, inherited instance variables are not shown. The default is No, which shows all instance variables defined for this flavor or inherited by this flavor.

:Sort {Alphabetical, Flavor} If Flavor, each instance variable is displayed along with the component flavor that provides it. The default is Alphabetical.

For example:

```
Command: Show Flavor Instance Variables CELL
     --> NEIGHBORS
         NEXT-STATUS
         STATUS
         X
         Y
```

Show Flavor Methods Command

Show Flavor Methods *flavor*

Displays all methods defined for the given flavor.

Keywords :Locally, :Match, :Output Destination, :Sort, and :Using Instance Variables. See the section "Keyword Options for Show Flavor Commands".

:Locally {Yes, No} If Yes, inherited methods are not shown. The default is No, which shows all methods defined for this flavor or inherited by this flavor.

:Match {*string*} Requests only those methods for generic functions that match the given string.

:Using Instance Variable
{*name*} Requests only those methods that use the instance variable named *name*.

Each line of output contains the name of the generic function, followed by the name of each flavor that provides a method for the generic function. If the method is not a primary method, its type is also displayed.

```
Command: Show Flavor Methods BOX-WITH-CELL
    -->  ALIVENESS method: CELL
         CHANGE-STATUS methods: CELL, BOX-WITH-CELL
         COUNT-LIVE-NEIGHBORS method: CELL
         :DESCRIBE method: FLAVOR:VANILLA
         others...
```

This command is similar to Show Flavor Operations. See the section "Show Flavor Operations **Command**". The difference between the two commands is in the perspective:

Show Flavor Methods displays information from an internal perspective, answering the question: What methods are defined for this flavor, or inherited from its component flavors?

Show Flavor Operations displays information from an external perspective, answering the question: What operations (generic functions and messages) are supported by instances of this flavor?

Show Flavor Operations Command

Show Flavor Operations *flavor keywords*

Shows all operations supported by instances of the given flavor, including generic functions and messages.

keywords	:Detailed, :Match, and :Output Destination. See the section "Keyword Options for Show Flavor Commands".
:Detailed	{Yes, No} If Yes, the display shows the arguments of each operation. The default is No.
:Match	{*string*} Shows only those operations matching the given substring.

For example:

```
Command: Show Flavor Operations BOX-WITH-CELL
    -->  ALIVENESS
         CHANGE-STATUS
         COUNT-LIVE-NEIGHBORS
         :DESCRIBE
         MAKE-INSTANCE
         SYS:PRINT-SELF (:PRINT-SELF)
         others...
```

One of the operations can be performed by using the generic function **sys:print-self** or sending the message **:print-self**. This operation was defined with **defgeneric**, using the **:compatible-message** option.

This command is similar to Show Flavor Methods. See the section "Show Flavor Methods **Command**". The difference between the two commands is in the perspective:

Show Flavor Operations displays information from an external perspective, answering the question: What operations (generic functions and messages) are supported by instances of this flavor?

Show Flavor Methods displays information from an internal perspective, answering the question: What methods are defined for this flavor, or inherited from its component flavors?

Show Generic Function Command

Show Generic Function *operation keywords*

Shows information on the given *operation*, which can be a generic function or message.

keywords	:Flavors, :Methods and :Output Destination. See the section "Keyword Options for Show Flavor Commands".
:Methods	{Yes, No} Yes displays all methods for the generic function, and their types.
:Flavors	{Yes, No} Yes displays the flavors that implement methods for the generic function.

For example:

```
Command: Show Generic Function CHANGE-STATUS
     --> Generic function CHANGE-STATUS takes arguments: (CELL-UNIT)
         This is an explicit DEFGENERIC in file SYS:EXAMPLES;FLAVOR-LIFE.
         Method-combination type is :AND :MOST-SPECIFIC-LAST.
```

Summary of Zmacs Commands for Flavors, Generic Functions, and Methods

This section lists the Zmacs commands that are related to flavors, generic functions and methods. In many cases the name of the command (and the use of the HELP key) is enough to begin using the command. The details of these commands are described elsewhere in the documentation.

Any tools that give information on ordinary functions can be applied to generic functions. The Zmacs commands listed below work for generic functions. See the section "Finding Out About Existing Code".

Quick Arglist c-sh-A
Show Documentation m-sh-D
Long Documentation c-sh-D
Function Apropos (m-X)
List Callers (m-X)
Multiple List Callers (m-X)
Edit Callers (m-X)
Multiple Edit Callers (m-X)

For documentation of the following Zmacs commands,see the section "Zmacs Commands for Flavors, Generic Functions, and Methods".

Edit Definition m-.
Show Effect of Definition (m-X)
Show Documentation Flavor m-sh-F
Quick Show Flavor Init Keywords and Documentation c-sh-F
Kill Definition (m-X)
Cleanup Flavor (m-X)
Add Patch Cleanup Flavor (m-X)
Insert Cleanup Flavor Forms (m-X)
List Methods (m-X)
Edit Methods (m-X)
List Combined Methods (m-X)
Edit Combined Methods (m-X)

The following Zmacs commands provide the same functionality as their command processor counterparts. For documentation of the CP commands, see the section "Show Flavor Commands".

There are two ways to enter the Zmacs Show Flavor commands: with or without a numeric argument of c-U. To enter a numeric argument, press c-U m-X before the command name. Without a numeric argument, you are prompted only for the required arguments. With a numeric argument, you are also prompted for keyword options. See the section "Keyword Options for Show Flavor Commands".

Show Flavor Components (m-X)
Show Flavor Dependents (m-X) *(see below)*
Show Flavor Differences (m-X)
Show Flavor Functions (m-X)
Show Flavor Handler (m-X)
Show Flavor Initializations (m-X)
Show Flavor Instance Variables (m-X)
Show Flavor Methods (m-X)
Show Flavor Operations (m-X)
Show Generic Function (m-X)

Show Flavor Dependents accepts numeric arguments other than c-U; the argument specifies how many levels of indirection to display. If you enter m-2 m-X Show Flavor Dependents, two levels of indirection are displayed.

Zmacs Commands for Flavors, Generic Functions, and Methods

Edit Definition m-.

This command is one of the most valuable tools of the system. When you are developing or debugging programs, you can use m-. to find the definition of an ordinary function, generic function, flavor, method, variable, package, or other type of definition. Completion is supported on the definition, if it is already in an editor buffer.

m-. prompts for a definition to find. You can enter a large variety of representations, and m-. figures out what definition you are seeking. For example, you can enter symbols with or without package prefixes.

You can provide any of the following responses to the m-. prompt:

symbol Finds the definition of *symbol*, which can be an ordinary function or generic function. For generic functions, the **defgeneric** form is found if one exists; all existing methods are also found. *symbol* can also be one of: variable, package, **defstruct** structure, flavor, or other types of definitions.

(*generic-function flavor*) Finds the definitions of one method that implements *generic-function* on instances of *flavor* and asks if you mean that method. If not, it proceeds to find other methods, including special-purpose methods such as **:before**, **:after**, **:default**, and so on.

(*symbol property*) Finds the function named by function spec (*:property symbol property*). This is a handy abbreviation.

function-spec Finds the definitions of *function-spec*. For example, you could enter **(flavor:method change-status cell)** to find the method of that function spec. Often it is more convenient to enter the list **(change-status cell)** instead.

When the requested Lisp object has multiple definitions, one of them is displayed. You can then use c-U m-. to cycle through the other definitions. Also, a list of all definitions and the files they are located in is stored in a buffer called **Definitions-n**. The position of the cursor in that buffer controls where c-U m-. will go next.

You can also point at forms with the mouse, in a buffer or in other windows, and click m-Left to edit the definition.

Show Effect of Definition (m-X)

Predicts the effect of evaluating the current definition (the definition the cursor is inside), or the current region, if a region is highlighted. **defflavor**, **defmethod**, **defgeneric**, **defwrapper**, and **defwhopper** forms are understood, among others.

Note: If the current definition is a method already defined in the world, this command predicts the effect of killing the definition of the method.

Show Documentation Flavor m-sh-F
Displays the documentation from the online documentation set for the current flavor. The current flavor is the flavor whose name the cursor is on, or the flavor marked by a region if a region is marked.

Quick Show Flavor Init Keywords and Documentation c-sh-F
Displays the initialization keywords for the current flavor. With a numeric argument, c-sh-F also shows the documentation string for the flavor.

Kill Definition (m-X)
Removes the current definition from the editor buffer and the world. If a patch is in progress, this also removes the definition from the world being patched. The current definition is the definition that the cursor is inside, or the definition marked by a region if a region is marked.

Cleanup Flavor (m-X)
Prompts for a flavor name. It then removes from the world any methods defined for that flavor that have been removed from the editor buffer for the file where they were defined.

Add Patch Cleanup Flavor (m-X)
Prompts for a flavor name. It then inserts **fundefine** forms in the current patch for any methods defined for that flavor that have been removed from the editor buffer for the file where they were defined.

Insert Cleanup Flavor Forms (m-X)
Prompts for a flavor name. It then inserts **fundefine** forms at the current point in the buffer for any methods defined for that flavor that have been removed from the editor buffer for the file where they were defined.

List Methods (m-X)
Prompts you for a generic function name (or message name). Lists the methods of all flavors that handle the generic function, in a mouse-sensitive display. To edit one of the methods, click on its function spec in the display. Alternatively, you can use c-. to edit a method. c-. cycles through the methods, each time choosing the next method for you to edit.

Edit Methods (m-X)
Prompts you for a generic function name (or message name). One of the definitions is found and pulled into an editor buffer. c-. cycles through the methods, each time choosing the next method for you to edit.

If more than one definition is available, the list of definitions and their source files is stored in a buffer *METHODS-*n*.

List Combined Methods (m-X)

Prompts for a generic function name, then for a flavor name. It then lists the methods for the specified generic function when applied to the specified flavor. This is a mouse-sensitive display. To edit one of the methods, click on its function spec in the display. Alternatively, you can use c-. to edit a method. c-. cycles through the methods, each time choosing the next method for you to edit.

Error messages appear if the flavor does not handle the generic function, or if the flavor requested is not a composed, instantiated flavor.

List Combined Methods (m-X) can be useful for predicting what a flavor will do in response to a generic function. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them.

Edit Combined Methods (m-X)

Asks you for a generic function name and a flavor name. This command finds all the methods that are called if that generic function is called on an instance of the given flavor. You can point to the generic function and flavor with the mouse; completion is available for the flavor name. As in Edit Methods (m-X), the command skips the display and proceeds directly to the editing phase.

Flavor Examiner

The Flavor Examiner enables you to examine flavors, methods, generic functions, and internal flavor functions defined in the Lisp environment. You can select the Flavor Examiner with SELECT X, or with the Select Activity Flavor Examiner command.

The Flavor Examiner lets you use all of the Show Flavor commands, saving the output in three history windows. Because much of the output is mouse-sensitive, it is convenient to use the mouse to select a flavor, method, or generic function from an output window to use as input to another Show Flavor command.

For a brief overview of the commands, see the section "Summary of Show Flavor Commands".

For a comprehensive description of each Show Flavor command, see the section "Show Flavor Commands".

Figure ! shows the initial window.

The Flavor Examiner window is divided into five panes.

Menu of Commands — the top-left pane

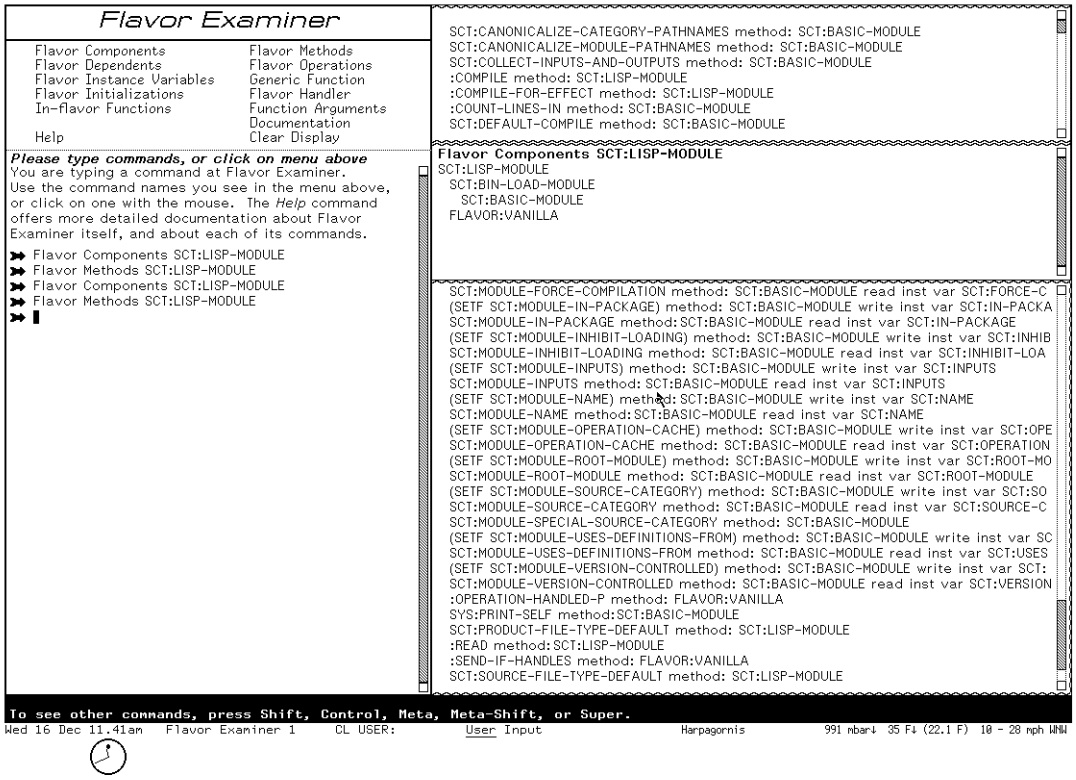


Figure 21. Flavor Examiner Window

The top-left pane offers a menu of flavor-related commands, such as `Flavor Components`; this is the same as the Show Flavor Components command. You can choose one of these commands by clicking Left or Right. Clicking Left makes the command appear in the Command Input Pane. Clicking Right makes the command appear and also displays the command's arguments, in a form that you can edit.

The Help command displays documentation on the flavor-related commands. The `HELP` key provides information on all the CP commands you can enter.

The Flavor Examiner offers two commands for clearing and refreshing the display. The `CLEAR DISPLAY` command clears the display from the three output panes; it first asks for confirmation. The `REFRESH DISPLAY` command displays the information on the screen again.

When you click Left or Right on a command name, the command appears in the Command Input Pane.

Command Input Pane — the bottom-left pane

The bottom-left pane is a command processor window. If you click on commands in the Menu of Commands, the commands appear in this window. You can enter argu-

ments (or commands) by typing them at the keyboard. This pane saves the history of all commands entered. You can click on the scroll bar to show different parts of the history.

You are not restricted to the commands in the Menu of Commands. You can give any command processor command.

The output of all commands appears in the Main Command Output Pane.

Main Command Output Pane — the bottom-right pane

Each command's output appears here. This pane saves the history of the output of all flavor-related commands. You can use the scroll bar to show different parts of the history.

Parts of the output of flavor-related commands are mouse-sensitive, so you can click on a flavor name or method name to enter it as an argument to another command.

If you give commands that are not flavor-related (such as the Show Host command), the output appears in a typeout window in the Main Command Output Pane. This kind of output is not saved in the history of this pane. The typeout window is itself a dynamic window with its own history.

When the output of the current command appears in the Main Command Output Pane, the output of the previous command is copied to the Previous Command Output Pane.

Previous Command Output Pane — the middle-right pane

This pane displays the output of the previous command. This pane does not save a history, but the second-to-last command is copied to the Second-to-Last Command Output Pane.

Second-to-last Command Output Pane — the top-right pane

This pane displays the output of the second-to-last command. This pane does not save a history. When another command is given, the contents of the Previous Command Output Pane are copied to this pane. Similarly, the contents of the Main Command Output Pane are copied to the Previous Command Output Pane.

Summary of Flavor Functions and Variables

This summary gives a brief description of all functions, macros, special forms, and variables related to Flavors. Each of the symbols listed here is described in detail elsewhere: See the section "Dictionary of Flavor Functions and Variables".

Basic Use Of Flavors

defflavor *name instance-variables component-flavors* &rest *options*
Defines a new flavor or redefines an existing one.

make-instance *flavor-name* &rest *init-options*
Creates and returns a new instance of the flavor.

defgeneric *name arglist* &body *options*
Defines a generic function, and enables you to specify options or documentation pertaining to a generic function as a whole.

defmethod
Defines a method that performs a generic function on objects of a given flavor.

compile-flavor-methods &rest *flavor-names*
Allows you to cause the combined methods of a program to be compiled at compile-time, and the data structures to be generated at load-time, rather than both happening at run-time.

Redefining Flavors, Instances, and Operations

change-instance-flavor *instance new-flavor*
Changes the flavor of an instance to another flavor.

recompile-flavor *flavor-name* &key *:generic :env :ignore-existing-methods (:do-dependents* **t***)*
Updates the internal data of the flavor and any flavors that depend on it.

flavor:remove-flavor *flavor-name*
Removes the definition of a flavor.

flavor:rename-instance-variable *flavor-name old new*
Changes the name of an instance variable, carrying the value of the old instance variable to the new for any existing instances.

flavor:transform-instance *instance new-fl*
Executes code when an instance is changed to a new flavor; thus enables you to perform initialization of the instance. Use this generic function by defining methods for it. It is not intended to be called.

Method Combination

define-simple-method-combination *name operator* &optional *single-arg-is-value (pretty-name* **(let ((*package* nil)) (format nil "~((~s)~)" flavor::name))***)*

Defines a new type of method combination that simply calls all the methods, passing the values they return to a given function.

define-method-combination *name parameters method-patterns* &body *body*
Enables you to declare a new type of method combination. Offers a rich declarative syntax.

The following tools are used in **define-method-combination** forms:

flavor:call-component-method *function-spec* &key *:apply (:arglist* **(if flavor::apply-p (prog1 (list apply) (setq apply t)) (prog1 flavor::*combined-method-arguments* (setq apply flavor::*combined-method-apply*)))***)*

Produces a form that calls the supplied function spec for a component method.

flavor:call-component-methods *function-spec-list* &key *(:operator* **'progn***)*
Produces a form that invokes the supplied function or special form. Each argument to that function is a call to one of the methods in the supplied list of function specs.

flavor:multiple-value-prog2 *before result* &rest *after*
Like **multiple-value-prog1** but returns all the values of the second form.

flavor:method-options *function-spec*
Extracts the method options portion of a method's function spec.

Internal Functions of Flavors

defun-in-flavor *(function-name flavor-name) arglist* &body *body*
Defines a function internal to a flavor.

defmacro-in-flavor *(function-name flavor-name) arglist* &body *body*
Defines a macro internal to a flavor.

defsubst-in-flavor *(function-name flavor-name) arglist* &body *body*
Defines a substitutable function internal to a flavor.

Wrappers and Whoppers

defwrapper *(generic-function flavor) (arglist* &rest *combined-method-body)* &body *body*
Defines a wrapper.

defwhopper *(generic-function flavor) arglist* &body *body*
Defines a whopper.

continue-whopper &rest *args*
Calls the methods for the generic function that was intercepted by the whopper. This is intended for use in **defwhopper** forms.

lexpr-continue-whopper &rest *args*
Like **continue-whopper**, but the last argument should be a list of arguments to be passed. This is useful when the arguments to the intercepted generic function include an **&rest** argument.

defwhopper-subst *(generic-function flavor) arglist* &body *body*
Defines a wrapper by combining the convenient syntax of **defwhopper** with the efficiency of **defwrapper**.

Variables

sys:*all-flavor-names*
A list of the names of all the flavors that have ever been created by **defflavor**.

flavor:*flavor-compile-trace-list*
A list of structures, each of which describes the compilation of a combined method into the run-time (not the compile-time) environment, in newest-first order.

self When a generic function is called on an object, the variable **self** is automatically bound to that object.

flavor:vanilla *and the Operations it Supports*

flavor:vanilla
The flavor included in all flavors that provides default behavior.

:describe
The object should print a description of itself onto the ***standard-output*** stream.

sys:print-self *object stream print-depth slashify-p*
The object should output its printed representation to the specified stream.

:print-self
This is a compatible message for **sys:print-self**.

:send-if-handles
The object should perform the operation (whether generic function or message) if it has a method for it.

:which-operations
The object should return a list of the messages and generic functions it can handle.

:operation-handled-p
The object should return **t** if it has a handler for the specified operation, **nil** if it does not.

get-handler-for *object operation*
Returns the method of the specified object for particular operation, or **nil** if the object has none.

Message-Passing

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.

send-if-handles *object message* &rest *arguments*
Sends a message to a flavor instance, if the flavor has a method defined for this message.

lexpr-send-if-handles *object message argument* &rest *arguments*
Like **send-if-handles**, except that the last element of arguments should be a list. All elements of that list are passed as arguments.

Note that **send-if-handles**, **:send-if-handles** and **lexpr-send-if-handles** work by sending the **:send-if-handles** message. You can customize the behavior of these operations by defining a method for that message.

A Flavor's Handler for an Operation

flavor:compose-handler *generic flavor-name* &key *:env*
Finds the methods that handle the specified generic operation on instances of the specified flavor.

flavor:compose-handler-source *generic flavor-name* &key *:env*
Finds the methods that handle the specified generic operation on instances of the specified flavor, and finds the source code of the combined method (if any).

operation-handled-p *object message-name*
Returns **t** if the flavor of the given instance has a method defined for the given generic function or message; **nil** otherwise.

get-handler-for *object operation*
Returns the method of the specified object for particular operation, or **nil** if the object has none.

A Flavor's default-init-plist

flavor:flavor-default-init-get *flavor indicator* &optional *default*
Retrieves a property from the default-init-plist of the specified flavor.

flavor:flavor-default-init-putprop *flavor value indicator*
Puts a property on the default-init-plist of the specified flavor.

flavor:flavor-default-init-remprop *flavor indicator*
Removes a property from the default-init-plist of the specified flavor.

A Flavor's Instance Variables

symbol-value-in-instance *instance symbol* &optional *no-error-p*
Allows you to read or write the value of an instance variable, or get a locative to an instance variable.

boundp-in-instance *instance symbol*
Returns **t** if the specified instance variable is bound in the given instance, **nil** otherwise.

Getting Other Information on Flavors

flavor:find-flavor *flavor-name* &optional *(error-p* **t***) env*
Informs you whether a given flavor is defined in the world.

flavor:flavor-allowed-init-keywords *flavor-name*
Provides a list of all symbols that are valid init options for a given flavor.

flavor-allows-init-keyword-p *flavor-name keyword*
Informs you whether a given keyword is a valid init option for the specified flavor.

flavor:get-all-flavor-components *flavor-name* &optional *env*
Returns a list of the components of the specified flavor, in the sorted ordering of flavor components.

Other Flavors Tools

instancep *x*
Returns **t** if the object is a flavor instance, **nil** otherwise.

flavor:describe-instance *instance*
Prints a description of an instance, including the values of its instance variables, to the ***standard-output*** stream.

flavor:print-flavor-compile-trace &key *:flavor :generic :newest :oldest :newest-first*
Enables you to view information on the compilation of combined methods that have been compiled into the run-time environment.

:unclaimed-message
If an operation is performed on a flavor instance, and no appropriate method exists, the Flavor system checks for a method for the **:unclaimed-message** message, and invokes it if it exists.

sys:property-list-mixin
This mixin flavor provides methods that perform a set of operations on property lists of flavors.

flavor:generic *generic-function-name*
Used in conjunction with the **:function** option for **defgeneric**.

sys:eval-in-instance *instance form*
Evaluates a form in the lexical environment of an instance.

sys:debug-instance *instance*
Enters the Debugger in the lexical environment of an instance.

flavor:with-instance-environment *(instance env)* &body *body*
Enables you to create a listener loop like that of the Debugger when examining a method, in which you can reference an instance's instance variables and internal functions directly.

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

zl:get-flavor-handler-for *flavor-name operation*
Given a flavor and an operation, this function returns the flavor's method for the operation or **nil** if it has none.

zl:locate-in-instance *instance symbol*
Returns a locative pointer to the cell inside the specified instance that holds the value of a specified instance variable.

zl:set-in-instance *instance symbol value*
Allows you to write the value of an instance variable.

zl:symeval-in-instance *instance symbol* &optional *no-error-p*
Allows you to read the value of an instance variable.

Method Combination

Before reading this section, it is important to understand the usual way the flavor system constructs a single handler from a set of available methods for a generic function. See the section "Inheritance of Methods".

This section describes how to use different types of method combination. The system offers several built-in types of method combination. You can also define your own type of method combination, using **define-simple-method-combination** or **define-method-combination**.

To use either a built-in type or a new type of method combination that you have defined, you supply the **:method-combination** option to **defflavor** or **defgeneric**. Often you also write special-purpose methods to be used in conjunction with the type of method combination, by supplying the *options* argument to **defmethod**.

Using the :method-combination Option

This section describes the syntax and use of the **:method-combination** option, which can be given to **defflavor** and **defgeneric**. Each generic function has an associated type of method combination. If no method combination is explicitly specified, the default is **(:method-combination :daemon :most-specific-first)**.

The syntax of this option is different for **defflavor** and **defgeneric**.

The **:method-combination** option to **defflavor** is given as follows:

```
(:method-combination
     generic-function name
     generic-function (name args...)
     ...)
```

The **:method-combination** option to **defgeneric** is given as follows:

```
(:method-combination name args...)
```

Each generic function is associated with a certain type of method combination specified by *name*. You can supply a built-in type of method combination, or a new type you have defined yourself. See the section "Built-in Types of Method Combination".

Sometimes the method combination type requires additional arguments, which are supplied as *args*.

In the built-in types of method combination, the first argument is usually the order that methods should be combined, either **:most-specific-first** or **:most-specific-last**. Often this argument is optional, and **:most-specific-first** is the default; this depends on the type of method combination used. See the section "**:most-specific-first** and **:most-specific-last** Method Order".

The following example uses **:daemon-with-or** method combination type for the generic function **fast-hardcopy**; the **:most-specific-first** method order is supplied. The **:case** method combination type is used for the generic function **set-attribute**.

```
(defflavor line-printer ((name "Tortoise")
                         (baud-rate 1200))
          ()
  :initable-instance-variables
  (:method-combination
        fast-hardcopy (:daemon-with-or :most-specific-first)
        set-attribute :case))
```

The following is an example of providing the **:method-combination** option to **defgeneric**:

```
(defgeneric hardcopy-file (device filename)
  (:method-combination :and :most-specific-first))
```

If **:method-combination** is specified for the same generic function by both **defflavor** and **defgeneric**, they must agree.

Any component of a flavor can specify the type of method combination to be used for a generic function. If more than one component of a flavor specifies a type of method combination, they must agree on the specification. Otherwise an error is signalled.

:most-specific-first and :most-specific-last Method Order

This section describes the meaning of **:most-specific-first** and **:most-specific-last**, two keywords often supplied as the *order* argument to the **:method-combination** option to **defflavor** and **defgeneric**.

In this example, the ordering of flavor components for **person** is:

```
(person primate living-thing flavor:vanilla)
```

person is the most specific flavor, and **flavor:vanilla** is the least specific flavor in the flavor ordering. **flavor:vanilla** is always the least specific flavor unless it is explicitly excluded from the flavor by using the **:no-vanilla-flavor** option to **defflavor**.

Several primary methods are available for the generic function **get-nutrition**:

- Flavor **person** supplies a method for **get-nutrition**, using a fork and spoon.
- Flavor **primate** supplies a method for **get-nutrition**, using hands (or paws) and teeth.

- Flavor **living-thing** supplies a very general method for **get-nutrition** that would be appropriate for any living thing, including plants.

- **flavor:vanilla** supplies no methods for **get-nutrition**.

Choosing a Single Primary Method

Some method combination types use the *order* argument to choose a single primary method from the set of available primary methods. **:daemon** method combination type is one example. If **:most-specific-first** order is used, the primary method chosen is the primary method that is supplied by the first flavor in the ordering of flavor components.

Using **:most-specific-first** method order, the method implemented by flavor **person** would be chosen.

Using **:most-specific-last** method order, the method implemented by flavor **living-thing** would be chosen.

Indicating the Order of Methods in a Combined Method

Most built-in types of method combination use the *order* argument to determine what order the methods are run in the combined method. For example, consider the **:or** type of method combination, which executes every method inside an **or** special form.

Using **:most-specific-first** order, the combined method resembles:

```
(or (method-for-person)
    (method-for-primate)
    (method-for-living-thing))
```

Using **:most-specific-last** order, the combined method resembles:

```
(or (method-for-living-thing)
    (method-for-primate)
    (method-for-person))
```

Built-in Types of Method Combination

You can use the following types of method combination by supplying one of the following keywords as the *name* argument to the **:method-combination** option to either **defgeneric** or **defflavor**.

The default method combination type **:daemon** is presented first. The other types appear in alphabetical order.

:daemon &optional (*order* '**:most-specific-first**)

This is the default type of method combination. **:daemon** selects all **:before** and **:after** methods, and a single primary method, which is the first method in the given *order*. The combined method calls the **:before** methods in

:most-specific-first order, then the chosen primary method, then the **:after** methods in **:most-specific-last** order. The result of the combined method is whatever the primary method returns. The combined method resembles:

```
(flavor:multiple-value-prog2
   (progn (before-method-1)
          (before-method-2))
   (primary-method-1)
   (progn (after-method-2)
          (after-method-1)))
```

:and &optional (*order* '**:most-specific-first**)
Selects all primary and **:and** methods and calls them in the given *order* inside an **and** special form. If a method returns **nil**, no more methods are called. The value returned is the value of the last method called, or **nil** if any method returns **nil**. Each **:and** method can return a single value only. The combined method resembles:

```
(and (method-1)
     (method-2)
     (method-3))
```

:append &optional (*order* '**:most-specific-first**)
Selects all primary and **:append** methods and calls them in the given *order* inside a call to the **append** function. Each of the methods must return a single value only, which is a list. The final result is the result of appending all these lists. The combined method resembles:

```
(append (method-1)
        (method-2)
        (method-3))
```

:case Invokes methods inside a **case** special form, using the second argument to the generic function to select the appropriate case. This argument is usually a keyword. Methods used with **:case** method combination must always include an *option* in their name, which specifies which case they implement. For example:

```
(defmethod
   (proceed subscript-out-of-bounds :new-subscript)
   (&optional (sub (prompt-and-read
                    :integer
                    "Subscript to use instead:")))
   "Supply a replacement subscript"
   (values :new-subscript sub))
```

This method is invoked by **(proceed *condition* :new-subscript)**, which prompts the user for a new subscript, and by **(proceed *condition* :new-subscript n)**, which uses **n** as the new subscript.

There are three special values of *option*, used to define special-purpose methods:

:otherwise This method is invoked if the second argument to the generic function is one for which a method has not been defined. The **:otherwise** method receives the unmatched second argument keyword as its first argument and the third and following arguments to the generic function as its remaining arguments. If no **:otherwise** method is defined for this flavor, a system-supplied method is called; it signals an error.

:which-operations This method is invoked if the second argument to the generic function is the symbol **:which-operations**. It is expected to take no additional arguments and to return two values: a list of the supported cases, and **t** if there is an otherwise method, or **nil** if there is not. If no **:which-operations** method is defined for this flavor, a system-supplied method is called.

:case-documentation
This method is invoked if the second argument to the generic function is the symbol **:case-documentation**. It is expected to take one argument (the third argument to the generic function), which is the name of one of the cases, and is expected to return a string, which is the documentation of that case, or **nil** if no documentation is known. If no **:case-documentation** method is defined for this flavor, a system-supplied method is called; it looks for documentation strings in the methods.

Note that the usual check for consistency of **defmethod** arguments with the arguments in the **defgeneric** form is not performed when **:case** method combination type is used, since the arguments can be different for each case.

Here is an example of defining a generic function that uses **:case** method combination type:

```
(defgeneric zzbuffer-next-unlinked-line (buffer selector line)
  "for buffers with disconnected sections, crosses a hard
   section boundary."
  (:method-combination :case)
  (:method (zznode :otherwise)
           (ignore selector line) nil) ; no unlinked lines
  (:method (zznode :foo)
           (ignore line) nil)          ; no selector here
  )
```

:daemon-with-and &optional (*order* '**:most-specific-first**)
Selects all **:before**, **:after**, and **:and** methods, and a single primary method. This is like the **:daemon** method combination type, except that the primary method is wrapped in an **and** special form after the **:and** methods. To write an **:and** method, supply the keyword **:and** as the *options* argument to

defmethod. The result is the values returned by the primary method if it is run, **nil** otherwise. The combined method resembles:

```
(flavor:multiple-value-prog2
   (progn (before-method-1)
          (before-method-2)
   (and (and-method-1)
        (and-method-2)
        (primary-method-1))
   (progn (after-method-2)
          (after-method-1)
```

The *order* argument indicates the order of **:and** methods and the choice of the primary method.

:daemon-with-or &optional (*order* '**:most-specific-first**)

Selects all **:before**, **:after**, and **:or** methods, and a single primary method. This is like the **:daemon** method combination type, except that the primary method is wrapped in an **or** special form with all the **:or** methods. To write an **:or** method, supply the keyword **:or** as the *options* argument to **defmethod**. The result is either the single value returned by the first **:or** method to return non-**nil**, or the values returned by the primary method (if it is run). The combined method resembles:

```
(flavor:multiple-value-prog2
   (progn (before-method-1)
          (before-method-2)
   (or (or-method-1)
       (or-method-2)
       (primary-method-1))
   (progn (after-method-2)
          (after-method-1)
```

The *order* argument indicates the order of **:or** methods and the choice of the primary method.

This is primarily useful for flavors in which a mixin introduces an alternative to the primary method. Each **:or** method gets a chance to run before the primary method and to decide whether or not the primary method should be run; if any **:or** method returns a non-**nil** value, the primary method is not run (nor are the rest of the **:or** methods).

:daemon-with-override &optional (*order* '**:most-specific-first**)

Selects all **:before**, **:after**, and **:override** methods, and a single primary method. This is similar to the **:daemon** and **:daemon-with-or** method combination types. The **:before** methods, the primary method, and the **:after** methods are run only if all of the **:override** methods return **nil**. The result is either the single value returned by the first **:override** method to return non-**nil**, or the values returned by the primary method (if it is run). To

write an **:override** method, supply the keyword **:override** as the *options* argument to **defmethod**. The combined method resembles:

```
(or (override-method-1)
    (override-method-2)
    (flavor:multiple-value-prog2
       (progn (before-method-1)
              (before-method-2))
       (primary-method-1)
       (progn (after-method-2)
              (after-method-1))))
```

The *order* argument indicates the order of **:override** methods and the choice of the primary method.

:inverse-list

Selects all primary and **:inverse-list** methods. The combined method calls each method in **:most-specific-first** with one argument; these arguments are successive elements of the list given as an argument to the generic function. Returns no particular value. If the result of a **:list**-combined generic function is sent back with an **:inverse-list**-combined generic function, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

The generic function is called as follows:

(*generic-function object arg1 arg2 arg3*)

The combined method resembles:

```
(progn (method-1 arg1)
       (method-2 arg2)
       (method-3 arg3))
```

:list &optional (*order* '**:most-specific-first**)

Selects primary and **:list** methods and calls them in the given *order* inside a call to the **list** function. Each method used with **:list** method combination can return a single value only. The result is a list of the returned values of the methods. The combined method resembles:

```
(list (method-1)
      (method-2)
      (method-3))
```

:max &optional (*order* '**:most-specific-first**)

Selects all primary and **:max** methods and calls them in the given *order* inside a call to the **max** function. Each method should return a numeric value. The result is the largest value of the set of values returned by the methods. The combined method resembles:

```
(max (method-1)
     (method-2)
     (method-3))
```

:min &optional (*order* '**:most-specific-first**)

Selects all primary and **:min** methods and calls them in the given *order* inside a call to the **min** function. Each method should return a numeric value. The result is the smallest value of the set of values returned by the methods. The combined method resembles:

```
(min (method-1)
     (method-2)
     (method-3))
```

:nconc &optional (*order* '**:most-specific-first**)

Selects all primary and **:nconc** methods and calls them in the given *order* inside a call to the **nconc** function. Each of the methods must return a single value only, which is a list. The final result is the result of concatenating these lists destructively. The combined method resembles:

```
(nconc (method-1)
       (method-2)
       (method-3))
```

:or &optional (*order* '**:most-specific-first**)

Selects all primary and **:or** methods and calls them in the given *order* inside an **or** special form. If a method returns a non-**nil** value, that value is returned and none of the other methods is called; otherwise, the next method is called. Thus each method is given a chance to handle the generic function. Methods that do not intend to handle the generic function should return **nil** to give the next method a chance to try. Each method can return a single value only. The combined method resembles:

```
(or (method-1)
    (method-2)
    (method-3))
```

:pass-on *order* **&rest** *arglist*

Selects all primary and **:pass-on** methods and calls them in the given *order*. The arguments to each method are the values returned by the preceding method. The values returned by the combined method are those the last method called. *arglist* is the argument list, which can include the **&aux** and **&rest** keywords.

For example, the flavors, generic function, and methods are defined as follows:

```
(defflavor f1 () ())
(defflavor f2 () (f1))
(defflavor f3 () (f2))

(defgeneric foo (fl a b c)
  (:method-combination :pass-on
   :most-specific-first a b c))
```

```
(defmethod (foo f1) (a b c) (values a b c))
(defmethod (foo f2) (a b c) (values a b c))
(defmethod (foo f3) (a b c) (values a b c))
```

The combined method for the generic function **foo** for flavor **f3** resembles:

```
(multiple-value-setq (A B C)
  (method-for-f3 A B C))
(multiple-value-setq (A B C)
  (method-for-f2 A B C))
(method-for-f1 A B C)
```

:progn &optional (*order* '**:most-specific-first**)
Selects primary and **:progn** methods and calls them in the given *order* inside a **progn** special form. The result of the combined method is whatever the last method returns. The combined method resembles:

```
(progn (method-1)
       (method-2)
       (method-3))
```

:sum &optional (*order* '**:most-specific-first**)
Selects all primary and **:sum** methods and calls them in the given *order* inside a call to the **+** function. The combined method resembles:

```
(+ (method-1)
   (method-2)
   (method-3))
```

:two-pass &optional (*order* '**:most-specific-first**)
Selects all primary methods and all **:after** methods. The combined method calls the primary methods in the given *order*, then the **:after** methods in **:most-specific-last** order. Returns the values returned by the last primary method that is executed. The combined method resembles:

```
(multiple-value-prog1
   (progn (primary-method-1)
          (primary-method-2)
          (primary-method-3))
   (progn (after-method-3)
          (after-method-2)
          (after-method-1)))
```

This is the type of method combination used by the **make-instance** generic function. For more information:

See the section "Writing Methods for **make-instance**".

Defining Special-Purpose Methods

This section describes how to define methods that are not primary methods; these are methods used in conjunction with different types of method combination.

:before and **:after** methods are examples of special-purpose methods; they are used with **:daemon** method combination type. For examples of defining **:before** and **:after** methods: See the section "Defining Before- and After-Daemons".

For information on which types or methods are appropriate for the various built-in method combination types: See the section "Types of Methods Used by Built-in Method Combination Types".

The syntax of **defmethod** is as follows:

```
(defmethod (generic-function flavor options..) (arg1 arg2..) body..)
```

The *options* argument is used when you are defining a special-purpose method. To define a primary method (the most common type of method), you do not supply the *options* argument.

To define an **:override** method (to be used with **:daemon-with-override** method combination):

```
(defmethod (update-display window-pane :override) (arg1 arg2..)
    body..)
```

To define a method to be used with **:case** method combination:

```
(defmethod (proceed subscript-out-of-bounds :new-subscript)
           (&optional
              (sub (prompt-and-read :integer
                                    "Subscript to use instead:")))
   "Supply a replacement subscript"
   (values :new-subscript sub))
```

Using :default Methods

A **:default** method is ignored if any primary methods are present. However, if no primary methods are defined, the **:default** method acts as a primary method.

This is useful in some types of method combination, such as **:and** and **:or**. That is, if no other methods are available to be combined in the **and** or **or** special form, the **:default** method is executed. The **:default** method might print a warning, or otherwise deal with the situation. If one or more primary methods are available, the **:default** method is not part of the combined method, and is not executed.

In the past, programmers defined **:default** methods for the most basic flavor. This usage guaranteed that the method would not be used in the combined method if a primary method were defined for any other flavor component. This was sometimes necessary, to compensate for the way that flavor components were ordered, in previous versions of the flavor system. It is no longer necessary to use **:default** methods for this purpose, because flavor component ordering works more predictably now than in previous releases.

Types of Methods Used by Built-in Method Combination Types

This section shows which types of methods are appropriate to be used in conjunction with each of the built-in types of method combination.

Some types of method combination expect a certain type of method to be used with them. For example, **:before** methods are used by **:daemon**, **:daemon-with-override**, **:daemon-with-or**, and **:daemon-with-and** method combination types. However, the other built-in types of method combination do not use **:before** methods.

It is important to realize that each method combination type ignores methods that are not appropriate for it. For example, if you have written a **:before** method for a generic function that uses **:list** method combination, that **:before** method will not be used in the combined method. A warning to that effect is printed.

Method Combination	Accepted Method Types
:and	primary, **:and**, **:default**
:append	primary, **:append**, **:default**
:case	:keyword, **:which-operations**, **:documentation**, **:otherwise**
:daemon	primary, **:before**, **:after**, **:default**
:daemon-with-and	primary, **:before**, **:after**, **:and**, **:default**
:daemon-with-or	primary, **:before**, **:after**, **:or**, **:default**
:daemon-with-override	primary, **:before**, **:after**, **:override**, **:default**
:inverse-list	primary, **:inverse-list**, **:default**
:list	primary, **:list**, **:default**
:max	primary, **:max**, **:default**
:min	primary, **:min**, **:default**
:nconc	primary, **:nconc**, **:default**
:sum	primary, **:sum**, **:default**
:or	primary, **:or**, **:default**

:pass-on	primary, **:pass-on**, **:default**
:progn	primary, **:progn**, **:default**
:two-pass	primary, **:after**, **:default**

The chart above shows that many types of method combination accept both primary methods and another kind of method. For example, **:and** method combination accepts primary and **:and** methods, and treats them the same when constructing the combined method. The only reason to use **:and** in the **defmethod** form is for documentation purposes; it clarifies that the method is intended to be used with **:and** method combination type. Other method types that fall into this category include: **:append**, **:or** when used with **:or** method combination, **:list**, **:max**, **:min**, **:inverse-list**, **:pass-on**, and **:progn**.

Example of Using Method Combination

This example might be part of a game in which each player has a country named by a color. Each country consists of a military presence, represented by the flavor **the-military**. Some countries have organized their military presence into branches represented by the flavors **army**, **navy**, and so on. In our example, **army** is built on **the-military**.

The game consists of moving armies and navies geographically. An important element of the game is keeping track of supplies, such as food and fuel. In this example, fuel is stored in three places:

- In a reserve storehouse — managed by **the-military** (it can be used by any branch of the military that needs it)
- In an auxiliary storehouse — managed by each branch
- In the field — managed by each branch

The following two **defflavor** forms represent **the-military** and **army**:

```
;;; the-military keeps its own storehouse of fuel supplies
;;; the-military has supplies of diesel, gas, and coal
(defflavor the-military ((reserve-diesel-supply 1200)
                         (reserve-gas-supply 1350)
                         (reserve-coal-supply 5600))
          ())
```

```
;;; the army's auxiliary supply augments fuel stored in the field
;;; the army has supplies of diesel and gas, but no coal
(defflavor army ((aux-diesel-supply 100)
                 (field-diesel-supply 150)
                 (aux-gas-supply 225)
                 (field-gas-supply 230))
          (the-military))            ;the-military is a component
```

Each player must keep track of fuel supplies. We organize this by writing a generic function **total-fuel-supply**. This function takes two arguments: *military-group* and *fuel-type*. It returns the total available supply of the given *fuel-type* (which can be **:diesel**, **:coal**, or **:gas**) for that military-group (it could be an instance of **army** or **the-military**). The generic function makes use of the **:sum** type of method combination.

```
;;; define a new type of method combination called :sum
;;; :sum creates combined methods like
;;;    (+ (method-1)
;;;       (method-2))
(define-simple-method-combination :sum + t)

(defgeneric total-fuel-supply (military-group fuel-type)
  "Returns today's total supply
    of the given type of fuel available to this military group."
  (:method-combination :sum :base-flavor-last))

(defmethod (total-fuel-supply army) (fuel-type)
  (case fuel-type
        (:diesel (+ aux-diesel-supply field-diesel-supply))
        (:gas (+ aux-gas-supply field-gas-supply))
        (:otherwise 0)))

(defmethod (total-fuel-supply the-military) (fuel-type)
  (case fuel-type
        (:diesel reserve-diesel-supply)
        (:gas reserve-gas-supply)
        (:coal reserve-coal-supply)
        (:otherwise 0)))
```

When we use **total-fuel-supply** on an instance of **the-military**, it responds with the amount of fuel available in the reserve storehouse managed by the-military. The handler is just the method for **the-military**.

When we use **total-fuel-supply** on an instance of **army**, it responds with the sum total of that type of fuel available to that object of **army** flavor, including its own auxiliary storehouse, its fuel in the field, and the fuel available to it stored in the reserve storehouse of the-military as a whole. The handler is a combined method that resembles:

```
(+ (method-for-army)
   (method-for-the-military))
```

Here is an illustration of using **total-fuel-supply**:

```
(setq blue-army (make-instance 'army))
=>#<ARMY 36431263>

(total-fuel-supply blue-army :coal)
=>5600

(total-fuel-supply blue-army :diesel)
=>1450
```

Defining a New Type of Method Combination

This section describes the tools that enable you to define a new type of method combination.

Summary of Method Combination Functions

define-simple-method-combination *name operator* &optional *single-arg-is-value (pretty-name* **(let ((*package* nil)) (format nil "~((~s)~)" flavor::name))***)*

Defines a new type of method combination that simply calls all the methods, passing the values they return to a given function.

define-method-combination *name parameters method-patterns* &body *body*
Enables you to declare a new type of method combination. Offers a rich declarative syntax.

The following tools are used in **define-method-combination** forms:

flavor:call-component-method *function-spec* &key *:apply (:arglist* **(if flavor::apply-p (prog1 (list apply) (setq apply t)) (prog1 flavor::*combined-method-arguments* (setq apply flavor::*combined-method-apply*)))***)*
Produces a form that calls the supplied function spec for a component method.

flavor:call-component-methods *function-spec-list* &key *(:operator* **'progn***)*
Produces a form that invokes the supplied function or special form. Each argument to that function is a call to one of the methods in the supplied list of function specs.

flavor:multiple-value-prog2 *before result* &rest *after*
Like **multiple-value-prog1** but returns all the values of the second form.

flavor:method-options *function-spec*
Extracts the method options portion of a method's function spec.

define-simple-method-combination *name operator* &optional *single-arg-is-value pretty-name* *Special Form*

Defines a new type of method combination that simply calls all the methods, passing the values they return to the function named *operator*.

It is also legal for *operator* to be the name of a special form. In this case, each subform is a call to a method. It is legal to use a lambda expression as *operator*.

name is the name of the method-combination type to be defined. It takes one optional parameter, the order of methods. The order can be either **:most-specific-first** (the default) or **:most-specific-last**.

When you use a new type of method combination defined by **define-simple-method-combination**, you can give the argument **:most-specific-first** or **:most-specific-last** to override the order that this type of method combination uses by default.

If *single-arg-is-value* is specified and not **nil**, and if there is exactly one method, it is called directly and *operator* is not called. For example, *single-arg-is-value* makes sense when *operator* is **+**.

pretty-name is a string that describes how to print method names concisely. It defaults to (**string-downcase** *name*).

Most of the simple types of built-in method combination are defined with **define-simple-method-combination**. For example:

```
(define-simple-method-combination :and and t)
(define-simple-method-combination :or or t)
(define-simple-method-combination :list list)
(define-simple-method-combination :progn progn t)
(define-simple-method-combination :append append t)
```

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

define-method-combination *name parameters method-patterns* &body *body* *Function*

Provides a rich declarative syntax for defining new types of method combination. This is more flexible and powerful than **define-simple-method-combination**.

name is a symbol that is the name of the new method combination type. *parameters* resembles the parameter list of a **defmacro**; it is matched against the parameters specified in the **:method-combination** option to **defgeneric** or **defflavor**.

method-patterns is a list of method pattern specifications. Each method pattern selects some subset of the available methods and binds a variable to a list of the function specs for these methods. Two of the method patterns select only a single method and bind the variable to the chosen method's function spec if a method is found and otherwise to **nil**. The variables bound by method patterns are lexically available while executing the *body* forms. See the section "*Method-Patterns* Option to **define-method-combination**". Each option is a list whose **car** is a keyword. These can be inserted in front of the body forms to select special options. See the

section "Options Available in **define-method-combination**". The *body* forms are evaluated to produce the body of a combined method. Thus the body forms of **define-method-combination** resemble the body forms of **defmacro**. Backquote is used in the same way. The *body* forms of **define-method-combination** usually produce a form that includes invocations of **flavor:call-component-method** and/or **flavor:call-component-methods**. These functions hide the implementation-dependent details of the calling of component methods by the combined method.

Flavors performs some optimizations on the combined method body. This makes it possible to write the body forms in a simple and easy-to-understand style, without being concerned about the efficiency of the generated code. For example, if a combined method chooses a single method and calls it and does nothing else, Flavors implements the called method as the handler rather than constructing a combined method. Flavors removes redundant invocations of **progn** and **multiple-value-prog1** and performs similar optimizations.

The variables **flavor:generic** and **flavor:flavor** are lexically available to the body forms. The values of both variables are symbols:

flavor:generic value is the name of the generic operation whose handler is being computed.

flavor:flavor value is the name of the flavor.

The *body* forms are permitted to **setq** the variables defined by the *method-patterns*, if further filtering of the available methods is required, beyond the filtering provided by the built-in filters of the *method-patterns* mechanism. It is rarely necessary to resort to this. Flavors assumes that the values of the variables defined by the method patterns (after evaluating the body forms) reflect the actual methods that will be called by the combined method body.

body forms must not signal errors. Signalling an error (such as a complaint about one of the available methods) would interfere with the use of flavor examining tools, which call the user-supplied method combination routine to study the structure of the erroneous flavor. If it is absolutely necessary to signal an error, the variable **flavor:error-p** is lexically available to the body forms; its value must be obeyed. If **nil**, errors should be ignored.

For a summary of all functions, macros, special forms, and variables related to Flavors, see the section "Summary of Flavor Functions and Variables".

The syntax of **define-method-combination** is complex. We present examples here and continue with the syntax later on in this section.

Examples of define-method-combination

This form defines the **:daemon** method-combination type:

```
(define-method-combination :daemon
                           (&optional (order ':most-specific-first))
          ;; select methods and bind them to variables
```

```
          ((before "before" :every :most-specific-first (:before))
           ;; select primary method,
           ;; if none is present, select :default method
           (primary "primary" :first order () :default)
           (after "after" :every :most-specific-last (:after)))
  ;;return values from primary method
  `(flavor:multiple-value-prog2
     ,(flavor:call-component-methods before)
     ,(flavor:call-component-method primary)
     ,(flavor:call-component-methods after)))
```

This form defines the **:two-pass** method combination type:

```
(define-method-combination :two-pass
                           (&optional (order ':most-specific-first))
          ((first "first-pass" :every order () :default)
           (second "after" :every :most-specific-last (:after)))
  ;;return values from last primary method to run
  `(multiple-value-prog1
     ,(flavor:call-component-methods first)
     ,(flavor:call-component-methods second)))
```

This form defines the **:inverse-list** method combination type:

```
(define-method-combination :inverse-list ()       ;take no parameters
        ;; select methods of type :inverse-list or :default
        ((methods "inverse-list" :every :most-specific-first
                                 () (:inverse-list) :default))
  (:arglist ignore list)
  (:method-transformer
    ;; each method receives a single argument, regardless
    (:generic-method-arglist `(list-element)))

  `(let ((list ,list))
     ,@(loop for (method . rest) on methods
           collect (flavor:call-component-method method
                    :arglist `(,(first list)))
           when rest
             collect `(setq list (cdr list)))))
```

This form defines the **:case** method combination type:

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: FLAVOR -*-

(define-method-combination :case (&optional
                                  (order ':most-specific-first))
        ((case-documentation "case-documentation" :first order
                                                  (:case-documentation))
         (which-operations "which-ops" :first order (:which-operations))
```

```
       (otherwise "otherwise" :first order (:otherwise))
       (cases "case ~s" :remove-duplicates order (*)))
(:order cases case-documentation which-operations otherwise)
(:arglist ignore subtype &rest args)
(:method-transformer
  ;; the cases methods receive funny arguments, and the arguments
  ;; may be different for each case, so don't bother validating them.
  (:inhibit-checking t))

(let ((alist (loop for case in cases
                   collect (list (first (method-options case)) case))))
  `(case ,subtype
     ,@(loop for (subtype method) in alist
             collect `((,subtype)
                       ,(call-component-method method :apply args)))
     ((:case-documentation)
      ,(if case-documentation
           (call-component-method case-documentation :apply args)
           `(get-case-documentation (first ,args) ',alist)))
     ((:which-operations)
      ,(if which-operations
           (call-component-method which-operations :apply args)
           `(values ',(mapcar #'car alist) ',(not (null otherwise)))))
     (otherwise
      ,(if otherwise
           (call-component-method otherwise)
           `(case-method-combination-missing-method self ',generic
                                                    ,subtype
                                                    ,args))))))
```

Method-Patterns **Option to define-method-combination**

Each *method-pattern* is a list of the form:

(*variable printer filter order pattern pattern...*)

variable is a variable to be bound to the list of method function specs resulting from the application of this *method-pattern* (or to a single function spec or **nil**, in the case where *filter* is one of the symbols **:first** or **:last**). Each *pattern* selects some subset of the methods (multiple patterns are combined with **or**). The *order* specifies the ordering of this subset and the *filter* specifies further pruning of it.

printer controls the concise printing of method function specs for Flavors tools. *printer* should concisely describe the options of the function spec; the generic function name and flavor are printed separately. When Flavors tools are used, *printer* is printed on ***standard-output***. Thus *printer* can be a string, which is given to **format** along with arguments that are the elements of the **flavor:method-options**

of the method's function spec. See the function **flavor:method-options**. Alternatively, *printer* can be a function to be called with *method-combination* and *function-spec* as its arguments. *printer* is not evaluated. The printing of **:default** methods is handled automatically, independent of *printer*.

filter is not evaluated; the filter type cannot be variable at run time. *filter* must be one of the following symbols:

:first Selects the first method in the specified order (**nil** if there are no methods).

:last Selects the last method in the specified order (**nil** if there are no methods).

:every Selects all the methods.

:remove-duplicates Selects all of the methods, excluding any duplicate methods. If duplicates are present, only the first method is selected. Duplicate methods are detected by applying **equal** to their **defmethod** *options*.

You can also do your own filtering in the body by using **setq** on a variable.

order is a form that must evaluate to **:most-specific-first** or **:most-specific-last**. Often it is simply one of those keywords, a self-evaluating constant. Another common practice is for *order* to be one of the variables in *parameters*.

Each *pattern* is a list or **nil**. A method matches the pattern if its *options* in the **defmethod** form match the given *pattern*. The pattern **nil** matches methods with no *options*; that is, primary methods. If you do not specify any patterns, a default pattern of **nil** is assumed. *pattern*s may be dotted lists.

Match is by **equal**, except that * matches anything. Each * in a pattern matches anything in that position of a method function spec. Dotted * might be useful for variable length.

You can intermix special symbols with the patterns. The only such symbol currently allowed is:

:default If the other patterns find any methods, this is ignored. If no other methods are found, the methods matched by the pattern **(:default)** are selected.

method-patterns are considered sequentially in the order they are written. The first *method-pattern* that matches a component method with its *patterns* takes care of that method, either including it or rejecting it depending on its *filter* and any **:default** processing specified. Subsequent *method-patterns* do not see that method. This means you should put the most general patterns last if more than one *method-pattern* clause could match the same method. The methods are expected to be called in the order the method patterns are written. If this is not so, you should include an **:order** clause. See the section "Options Available in **define-method-combination**".

Any methods not taken care of by any method patterns are extraneous, and Flavors warns about them. Flavors automatically takes care of special methods, such as wrappers and whoppers; you need not do anything to handle them when defining a new method combination type.

Here is an example of the *method-patterns* used by **:case** method combination type:

```
((case-documentation "case-documentation" :first order (:case-documentation))
 (which-operations "which-operations" :first order (:which-operations))
 (otherwise "otherwise" :first order (:otherwise))
 (cases "case ~s" :remove-duplicates order (*)))
```

Options Available in define-method-combination

The *options* to **define-method-combination** include:

(**:arglist** *args...*)
: Specifies the argument list of the generic function, including the first or **self** argument. You can use **ignore** for arguments of no interest to the method-combination function, such as the **self** argument. As the *body* forms are being executed, each variable in *args* is bound to a form that accesses the relevant argument when evaluated as part of the combined-method body returned by the *body* forms. &optional and &rest are permitted in *args*. If **:dispatch** is used in the **defgeneric** form, the **:arglist** option mentions the arguments in the order received by the methods, with the dispatching argument moved to the front.

  The **:arglist** option is useful when the action of the combined-method depends on the arguments (rather than simply passing the arguments on to the component methods), as in **:case** or **:inverse-list** method combination. See the section "Examples of **define-method-combination**".

(**:order** *vars...*)
: Specifies the order in which the component methods will be called, for the benefit of flavor examining tools. Each of the *vars* must be a variable bound by one of the *method-patterns*. Normally all of the *vars* bound by *method-patterns* would be included in the **:order** option, but this is not required. If no order is specified, the default is to use the *vars* in the order in which the *method-patterns* are written.

(**:method-transformer** *clauses...*)
: This can make changes to the arguments and body of methods for generic functions that use this type of method combination, and can control the validation of the arguments in the **defmethod** against the arguments in the **defgeneric**. Code in *clauses* receives arguments bound to the following variables, and also has the parameters of the **define-method-combination** available to it:

sys:function-spec
The name of the method

flavor:method-combination
The method-combination type

flavor:method-arglist
The arglist specified for the method

flavor:method-body
The body of the method, including declarations

flavor:generic-method-arglist
The arglist specified for methods in the **defgeneric**.

Each *clause* consists of a keyword and a form evaluated to produce a replacement for a normal item. The clauses are:

:method-arglist
Replaces the argument list specified in **defmethod**.

:method-body
Replaces the body specified in **defmethod**.

:generic-method-arglist
Replaces the argument list for methods specified in **defgeneric**.

:inhibit-checking
If non-**nil**, inhibits comparing **flavor:method-arglist** and **flavor:generic-method-arglist** for the wrong number of arguments.

Interface to the Method Combination System

The interface takes care of calling the functions defined by the expansion of the **define-method-combination** special form, optimizing the results, and adding wrappers and whoppers to it to make the complete combined method.

The external interface to the method combination system consists of the following functions:

flavor:compose-handler *generic flavor-name* &key *:env*
Finds the methods that handle the specified generic operation on instances of the specified flavor.

flavor:compose-handler-source *generic flavor-name* &key *:env*
Finds the methods that handle the specified generic operation on instances of the specified flavor, and finds the source code of the combined method (if any).

Defining Functions Internal to Flavors

Flavors allows you to define functions that are lexically "inside" the flavor. Such functions have access to the instance variables of some instance of that flavor, or of a flavor built on it, defined by the value of **self**. There are two kinds of functions inside a flavor: methods and internal functions.

Methods	*Internal Functions*
Can be called from anywhere, via the generic function mechanism.	Can be called only from another function already inside the flavor. The caller can be a method or an internal function.
Can serve as entry-points or interfaces.	Can be used to share subroutines among methods.
Receive **self** as an argument.	Share the **self** of their caller.
Are defined with **defmethod**.	Are defined with **defun-in-flavor**, **defsubst-in-flavor**, and **defmacro-in-flavor**.
Have lexical access to instance variables of the object **self**.	Have lexical access to instance variables of the object **self**.
Are inherited from component flavors, using method combination rules.	Are inherited from component flavors, somewhat differently.

You can shadow a globally defined function with an internal function with the same name. The globally defined function can be an ordinary function or a generic function. However, common practice is to use unique names for internal functions, to minimize possible confusion.

Scoping of Internal Functions

The scoping of internal functions is as if each function inside the flavor were surrounded by a **flet** form that declared all of the internal functions. This is analogous to the scoping of instance variables. Internal functions are inherited from component flavors. If there are several internal functions with the same name, the first one in the ordering of flavor components is chosen.

If the name of an internal function of a flavor is used with the **function** special form (or the #' syntax), a closure is created that captures the value of **self** and the instance variables. This closure can be passed as a functional argument. Note that you use the name of the internal function, not its function spec.

Because internal functions of flavors are lexically scoped, they must be declared or defined before their callers. Otherwise the caller would not be compiled or evaluated in the proper lexical environment, and would not know that the name of the internal function refers to an internal function rather than to an ordinary function in the global environment. The **:functions** option to **defflavor** can be used to declare the names of **defun-in-flavor** internal functions of the flavor.

Inheritance of Internal Functions

The inheritance of internal functions works differently than the inheritance of methods. The following example suggests that it is good practice not to give two internal functions for related flavors the same name:

```
(defflavor flav1 () ())
(defflavor flav2 () (flav1))
(defmethod (meth flav1) () 1)
(defmethod (meth flav2) () 2)
(defun-in-flavor (func flav1) () 1)

(defmethod (test flav2) () (list (meth self) (func)))

(test (make-instance 'flav2))
=> (2 1)
    ;method calls (defun-in-flavor func flav1)

;;;now we define a new defun-in-flavor for flav2
(defun-in-flavor (func flav2) () 2)

(test (make-instance 'flav2))
=> (2 1)
    ;method still calls (defun-in-flavor func flav1),
    ;even though there's now a (defun-in-flavor func flav2)
```

The last example shows that when you define an internal flavor function that shadows another internal flavor function for a less specific flavor, any methods written on the more specific flavor are still left calling the less specific function.

In the method for **test**, the inheritance of **meth** depends on the actual flavor of **self** at run-time, but the inheritance of **func** depends on the flavor for which the method that calls **func** is being defined, **flav1** in this case. Also, the binding from the short name of an internal flavor function to the full function-spec happens at compile time, not at the time the function is called. If you want the more dynamic inheritance, use methods instead of internal flavor functions.

Redefining Internal Functions

You can redefine internal flavor functions by evaluating another **defun-in-flavor**, **defsubst-in-flavor**, or **defmacro-in-flavor** form. The new definition replaces the old.

Note that if you have defined an internal function, and decide to change it to a method, you must remove the definition of the internal function using **fundefine** and the function spec of the internal function.

Function Specs for Internal Functions

For documentation on the function specs for internal functions, macros, and substitutable functions, see the section "Function Specs for Flavor Functions".

Related Functions:

defun-in-flavor *(function-name flavor-name) arglist* &body *body*
Defines a function internal to a flavor.

defmacro-in-flavor *(function-name flavor-name) arglist* &body *body*
Defines a macro internal to a flavor.

defsubst-in-flavor *(function-name flavor-name) arglist* &body *body*
Defines a substitutable function internal to a flavor.

Wrappers and Whoppers

Wrappers and whoppers are used in certain cases in which **:before** and **:after** daemons are not powerful enough. **:before** and **:after** daemons let you put some code before or after the execution of a method; wrappers and whoppers let you put some code *around* the execution of the method. For example, you might want to bind a special variable to some value during the execution of a method. Or you might want to establish a condition handler, or set up a **catch** or **unwind-protect**. Wrappers and whoppers can also decide whether or not the primary method should be executed.

Whoppers are used more frequently than wrappers.

Wrappers	*Whoppers*
Similar to a macro.	Similar to a function.
If a wrapper is modified, all combined methods using it must be recompiled (this is done automatically).	If a whopper is modified, only the whopper must be recompiled.
The body of a wrapper is expanded in all the combined methods in which it is involved. The code is duplicated, not shared.	The body of a whopper is not expanded in multiple places.
Wrappers are slightly faster than whoppers.	Whoppers require two extra function calls each time

they are called.

Because they involve the interaction of several complex mechanisms, you should use great care when using wrappers and whoppers.

The function specs for wrappers and whoppers are described elsewhere: See the section "Function Specs for Flavor Functions".

Changing and removing the definition of wrappers and whoppers is described elsewhere: See the section "Redefining Flavors, Methods, and Generic Functions".

Summary of Functions Related to Wrappers and Whoppers

defwrapper *(generic-function flavor) (arglist* &rest *combined-method-body)* &body *body*
Defines a wrapper.

defwhopper *(generic-function flavor) arglist* &body *body*
Defines a whopper.

continue-whopper &rest *args*
Calls the methods for the generic function that was intercepted by the whopper. This is intended for use in **defwhopper** forms.

lexpr-continue-whopper &rest *args*
Like **continue-whopper**, but the last argument should be a list of arguments to be passed. This is useful when the arguments to the intercepted generic function include an **&rest** argument.

defwhopper-subst *(generic-function flavor) arglist* &body *body*
Defines a wrapper by combining the convenient syntax of **defwhopper** with the efficiency of **defwrapper**.

Examples of Wrappers

The use of **defwrapper** is best explained by example. Suppose you need a lock locked during the processing of the **drain** generic function on an instance of the **cistern** flavor. The **drain** function takes one argument, *valve-position*. You have written a **lock-faucet** special form that knows how to lock the lock. **lock-faucet** needs to know the valve-position, the first argument to the **drain** function.

```
(defwrapper (drain cistern) ((valve-position) form)
  ;;; set lock for duration of method
  `(lock-faucet (self valve-position)
     ,form))                          ; Execute method itself
```

Note that the argument variable **valve-position** is not referenced with a comma preceding it. Argument variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done. Rather, the result of that macroexpansion and back-quoting is a form that is evaluated with those variables

bound to the arguments of the generic function, at the time the generic function is called.

Consider another example. You might want to run some code before executing the primary method. In addition, if the argument is **nil** you wanted to return from the generic function immediately, without executing the primary method. You could not do this using a **:before** daemon because **:before** methods are constrained to proceed to the primary method. You can use a wrapper to solve the problem. The following wrapper checks the argument and prevents anything further from happening, if it is **nil**. Instead of having a **:before** daemon, the "before" code is incorporated into the wrapper itself:

```
(defwrapper (drain cistern) ((valve-position) form)
  `(unless (null valve-position)  ; Do nothing if valve closed
     before-code                  ; Execute some "before" code
     ,form))                      ; Execute the body of the method
```

Suppose you need a variable for communication among the daemons for a particular generic function; perhaps the **:after** daemons need to know what the primary method did, and it is something that cannot be easily deduced from the arguments alone. You might use an instance variable for this, or you might create a special variable that is bound during the processing of the generic function, and used free by the methods. For example:

```
(defvar *communication*)
(defwrapper (drain cistern) (ignore form)
  `(let ((*communication* nil))
     ,form))
```

Similarly you might want a wrapper that puts a **catch** around the processing of a generic function so that any one of the methods could throw out in the event of an unexpected condition.

Examples of Whoppers

The following whopper adds code around the execution of the method that performs the generic function **print-integer** on instances of the **foo** flavor. Specifically, the whopper binds the value of the special variable **base** to 3 around the execution of the method. This function takes one argument, *n*:

```
(defwhopper (print-integer foo) (n)
  (let ((base 3))
    (continue-whopper n)))
```

The following whopper sets up a **catch** around the execution of the **compute-height** method of flavor **giant**, regardless of what arguments this methods accepts:

```
(defwhopper (compute-height giant) (&rest args)
  (catch 'too-high
     (lexpr-continue-whopper args)))
```

Mixing Flavors that Use Whoppers and Daemons

Like daemon methods, whoppers work in outward-in order; when you add a **defwhopper** to a flavor built on other flavors, the new whopper is placed outside any whoppers of the component flavors. However, *all* whoppers happen before *any* daemons happen. Thus, if a component defines a whopper, methods added by new flavors are considered part of the continuation of that whopper and are called only when the whopper calls its continuation.

Mixing Flavors that Use Wrappers and Whoppers

Whoppers and wrappers are considered equal for purposes of combination. If two flavors are combined, one having a wrapper and the other having a whopper for some method, then the wrapper or whopper of the flavor that is further out is on the outside. If, for some reason, the very same flavor has both a wrapper and a whopper for the same message, the wrapper goes outside the whopper.

Mixing Flavors that Use Wrappers and Daemons

When mixing flavors, wrappers work in outside-in order, just as daemons work. When you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. When the combined method is built, the calls to the before daemon methods, primary methods, and after daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

Complete Options for defflavor

This section describes the options to **defflavor**. For information on the format of the options, see the special form **defflavor**.

See the section "Inheritance of **defflavor** Options".

:abstract-flavor

Declares that the flavor exists only to define a protocol; it is not intended to be instantiated by itself. Instead, it is intended to have more specialized flavors mixed in before being instantiated.

Trying to instantiate an abstract flavor signals an error.

:abstract-flavor is an advanced feature that affects paging. It decreases paging and usage of virtual memory by allowing abstract flavors to have combined methods. Normally, only instantiated flavors get combined methods, which are small Lisp functions that are automatically built and compiled by the flavor system to call all of the methods that are being combined to make the effective method. Sometimes many different instantiated flavors use the same combination of methods. If this is the case, and the abstract flavor's combined methods are the same ones that

are needed by the instantiated flavors, then all instantiated flavors can simply share the combined methods of the abstract flavor instead of having to each make their own. This sharing improves performance because it reduces the working set.

compile-flavor-methods is permitted on an abstract flavor. It is useful for combined methods that most specializations of that flavor would be able to share.

(**:area-keyword** *symbol*)

Changes the **:area** keyword to **make-instance** of this flavor to the supplied argument. This is useful if the flavor is using the **:area** keyword for some other purpose, such as an init keyword for an object's geometric or geographic area. This option is similar to the **:conc-name** option. Whereas **:conc-name** enables you to supply a prefix for an automatically generated function, the **:area-keyword** option to **defflavor** enables you to rename the **:area** keyword that is given to **make-instance** when making an instance of this flavor.

In rare cases, you might want to use the keyword **:area** for other purposes. For example, a geometric program might have a **triangle** flavor that has an instance variable named **area** that is equal to one-half of the product of its width and height. It is then important to distinguish between the instance variable **area** and the keyword that denotes in which area to create instances. You can rename that keyword for the affected flavor by providing the **:area-keyword** keyword to **defflavor**. For example:

```
(defflavor triangle (area)    ;instance var.
           ()
  :initable-instance-variables
  (:area-keyword :cons-area))
```

Now, when you make an instance, **:area** refers to the instance variable, and **:cons-area** indicates the area in which instances are to be created:

```
(make-instance 'triangle
                  :area 44.23
                  :cons-area GEOMETRY-AREA)
```

(**:component-order** *args...*)

Enables you to state explicitly the ordering constraints for the flavor components whose order is important. You can use it to relax ordering constraints on component flavors for which order is not important. You can also use it to add ordering constraints on flavors that are not components; this means that if this flavor is later mixed with another flavor, the ordering of components takes into account the constraints given by this option.

If **:component-order** is given, the order of flavor components at the top of the **defflavor** form is no longer significant. The arguments to

:component-order are lists. The members of each list are constrained to appear in the order they appear in the list. Any component that does not appear in these lists has no ordering constraints placed on it.

For example, the following form imposes many constraints on the ordering of the seven flavor components:

```
(defflavor foo (var1 var2)
          (a b c d e f g))
```

However, your program might not depend on a specific ordering of components, because the components have no effect on each other. For example, your program might depend only these ordering constraints:

- Flavor **c** must precede **d**.
- Flavor **b** must precede **g**.
- If flavor **x** is present (it could be a component of one of the components, or it could be mixed into flavor **foo** to create a new flavor), it must follow **b** and precede **g**.

You can specify those restrictions by giving the following option to **defflavor**:

```
(defflavor foo (var1 var2)
          (a b c d e f g)
  (:component-order (c d) (b x g)))
```

Note that this does not constrain flavors **c** and **d** to precede flavors **b**, **x**, and **g**. Also, it is not an error to specify an ordering constraint for a flavor that is not a component of this flavor. For example, it is valid to constrain the order of **x**, although **x** might not be a direct or indirect component of this flavor.

For details on the way the flavor system determines the order of flavor components, see the section "Ordering Flavor Components".

(**:conc-name** *symbol*)

Enables you to specify a prefix for the accessor functions created when the **:readable-instance-variables**, **:writable-instance-variables**, or **:locatable-instance-variables** options are supplied. Normally the accessor function to access the instance variable *v* of flavor *f* is named *f-v*, such as:

```
ship-mass
```

You can specify a different prefix as follows:

```
(defflavor ship (captain mass) ()
  (:conc-name get-)
  :readable-instance-variables)
```

The accessor functions for **ship** are thus named **get-captain** and **get-mass**.

You can use **:conc-name** to specify no prefix as follows:

```
(defflavor ship (captain mass) ()
  (:conc-name nil)
  :readable-instance-variables)
```

The accessor functions for **ship** are named **captain** and **mass**.

For information on an alternate syntax of **defflavor** which enables you to explicitly specify the names of accessor functions, see the section "Specifying Names for Functions that Access Instance Variables".

(**:constructor** *args...*)

Automatically generates a constructor function that enables you to create new instances of this flavor. The main advantage to constructor functions is that they are much faster than using **make-instance**. Whereas **make-instance** takes a flavor name argument and looks up information about the flavor, constructor functions have this information compiled into Lisp code. Constructor functions run any **make-instance** methods that are defined for this flavor.

The constructor function can take positional arguments instead of keyword arguments, or a mix of positional and keyword arguments. You can give the **:constructor** option more than once within a single **defflavor** to define several different constructors, each with its own arguments.

It is necessary to do a **compile-flavor-methods** to ensure that the constructor function is defined. The constructor function is defined as part of the macro-expansion of **compile-flavor-methods**, not part of the macro-expansion of **defflavor**. This makes it possible to define a flavor before all component flavors are defined. If you are incrementally developing code, you can put a **compile-flavor-methods** form into an editor buffer and use `c-sh-C` before running code that calls the constructor function.

The **:constructor** option takes two arguments. The first argument specifies the name of the constructor function. The second argument is optional; it is used to describe the argument list of the constructor. If no second argument is given, the constructor function uses the same **&key** arglist that **make-instance** would accept after the flavor name, including keywords for: each initable instance variable, each init-keyword, and **:area**.

For example, a simple case like **(:constructor make-foo (a b c))** defines **make-foo** to be a three-argument constructor function whose arguments are used to initialize the instance variables named **a**, **b**, and **c**.

In addition, you can use the keywords **&optional**, **&rest**, **&key**, **&allow-other-keys**, and **&aux** in the argument list. They work as you might expect, but note the following:

```
(defflavor foo (a b c d e f g h i) ()
  (:constructor make-foo
     (a &optional b (c 'sea)
        &rest d
        &key i
        &aux e (f 'eff))))

(compile-flavor-methods foo)
```

This defines **make-foo** to be a constructor that accepts one or more arguments. The first argument is used to initialize the **a** instance variable. The second argument is used to initialize the **b** instance variable. If there is no second argument, then the default value given in the body of the **defflavor** (if given) is used instead. The third argument is used to initialize the **c** instance variable. If there is no third argument, then the symbol **sea** is used instead. Any arguments following the third argument are collected into a list and used to initialize the **d** instance variable. If the keyword **:i** is supplied with a value, that value is used to initialize the **i** instance variable. If there are three or fewer arguments, then **nil** is placed in the **d** instance variable; the initial value of **e** is undefined; and the **f** instance variable is initialized to contain the symbol **eff**.

The actions taken in the **b** and **e** cases were carefully chosen to allow you to specify all possible behaviors. Note that the **&aux** "variables" can be used to completely override the default initializations given in the **defflavor** form.

In summary, each parameter of a constructor can be:

- The name of an instance variable. In this case, the argument is stored into the instance variable.

- Something that matches the area keyword; by default this is **:area**, but the **:area-keyword** option can be used to specify a different area keyword. In this case, the argument is the area in which the instance is created.

- Something that matches an init keyword declared by the **:init-keyword** option. In this case, the argument is passed to the **make-instance** and/or **:init** methods as the appropriate keyword argument.

Above, "something that matches" means the following: For **&key** parameters, something matches if it is **eq** to the keyword. Note the **(&key ((:key** *var*)**))** syntax can be used to specify a keyword that is different from the variable name. For non-**&key** parameters, something matches if it is **eq** to a symbol with the same name as the parameter variable, but in the keyword package.

Defaulting occurs as follows: If no default is specified in the constructor argument list, then the default is not **nil** (unlike the behavior of a nor-

mal argument list). Instead, for **&aux** parameters the default is to leave the instance variable unbound or not pass the init keyword to the **make-instance** and/or **:init** methods. For non-**&aux** parameters, if no default is specified then default is whatever default that **make-instance** would use. The area defaults the same regardless of whether it is **&aux** or not, since it is impossible not to have an area for making an instance.

(**:default-handler** *function-name*)

The argument is the name of a function that is to be called when a generic function is called for which there is no method. The function is called with the arguments the instance was called with, including the message name; the values it returns are returned. If this option is not specified on any component flavor, it defaults to a function that signals an error.

The function specified with the **:default-handler** option to **defflavor** receives two additional arguments. The first argument is **self** and the second is always **nil**.

The following example shows the use of **:default-handler**.

```
(defflavor lisp-stream (forward) ()
  (:default-handler lisp-stream-forward))

(defun lisp-stream-forward (self ignore message &rest arguments)
  (lexpr-funcall (send self :forward) message arguments))
```

This is equivalent to defining a method for the **:unclaimed-message** message. See the message **:unclaimed-message**.

(**:default-init-plist** *plist*)

Provides a list of alternating keywords and default value forms for keyword arguments that are allowed for **make-instance** of this flavor. The keywords can be: initable instance variables, init keywords, required init keywords, **:allow-other-keys**, or **:area**. If you do not specify one or more of these keywords as arguments to **make-instance**, they are set to the default in **:default-init-plist**. For the meaning of the **:allow-other-keys** and **:area** keywords, see the function **make-instance**.

This option allows one component flavor to default an option to another component flavor. The value forms are evaluated only when and if they are used. For example, the following would provide a default "frob array" for any instance for which the user did not provide one explicitly as an argument to **make-instance**:

```
(:default-init-plist :frob-array
                     (make-array 100))
```

Elements in the **:default-init-plist** that are init keywords are supplied to any methods defined for **make-instance** of this flavor. However, **:default-init-plist** elements that initialize instance variables are not supplied to **make-instance** methods.

(**:documentation** *string*)

The list of arguments to this option is remembered on the flavor's property list as the **:documentation** property. The convention for this list is as follows: A string describes what the flavor is for; this could consist of a brief overview in the first line followed by several paragraphs of detailed documentation. A symbol is one of the following keywords:

:mixin	A flavor that you might want to mix with others to provide a useful feature.
:essential-mixin	A flavor that must be mixed in to all flavors of its class, or inappropriate behavior follows.
:lowlevel-mixin	A mixin used only to build other mixins.
:combination	A combination of flavors for a specific purpose.
:special-purpose	A flavor used for some internal purpose by a particular program, which is not intended for general use.

(**:functions** *internal-function-names*)

Declares the names of **defun-in-flavor** functions internal to the flavor. The arguments are names of internal functions.

defun-in-flavor functions are lexically scoped. They must either be defined before their callers, or declared by using the **:functions** option for **defflavor**. Otherwise the caller would not be compiled or evaluated in the proper lexical environment, and would not know that the name of the internal function refers to a **defun-in-flavor** function rather than to an ordinary function in the global environment. see the section "Defining Functions Internal to Flavors".

The **:functions** option is not appropriate for internal functions defined by **defsubst-in-flavor** or **defmacro-in-flavor**.

(**:gettable-instance-variables** *vars...*)

This option is available for compatibility with the old message-passing flavor system. When writing new code, it is good practice to use the **:readable-instance-variables** option instead.

Enables automatic generation of methods for getting the values of instance variables. The message name is the name of the variable, in the keyword package (that is, put a colon in front of it.)

To use the automatically-generated method, use the old **send** syntax. For example:

```
(send my-ship :mass)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

(**:init-keywords** *symbols...*)

Declares its arguments as keywords that should be accepted by **make-instance** of this flavor. Init keywords are:

- Keyword arguments to be processed by methods defined for **make-instance** for this flavor

  When you write a method for **make-instance** that accepts keyword arguments, you should include those keywords as init keywords. For related information, see the section "Writing Methods for **make-instance**".

- Keywords defined by the **:mixture** option to **defflavor**

  When you define a family of flavors, you should include any **:mixture** keywords as init keywords.

:init-keywords is used for error-checking when you create an instance. All keywords given to **make-instance** are either initable instance variables or init keywords, or keywords accepted by **make-instance**, like **:area**. If the caller misspells a keyword or otherwise uses a keyword that no component flavor handles, **make-instance** signals an error.

Note that whenever you have a **:required-init-keywords** clause containing keywords that are to be used by **make-instance** methods, it is necessary to include those keywords in the **:init-keywords** clause as well.

It is allowed but pointless to include keywords that initialize instance variables as init keywords.

(**:initable-instance-variables** *vars...*)

Enables you to initialize the specified instance variables when making an instance of this flavor. The instance variables are sometimes called *initable*. In the **make-instance** form, you initialize values by including the keyword (name of the instance variable) followed by its value, as follows:

```
(make-instance flavor-name :variable-name value)
```

For example, when making an instance of the **ship** flavor, we can initialize the **mass** instance variable (assuming it is initable), as follows:

```
(make-instance 'ship :mass 105)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

If you want to specify that an instance variable inherited from another component flavor should be initable, you can include the name of that instance variable explicitly in the top of the **defflavor** form. This option checks for spelling errors, so any instance variables that are declared

initable must appear at the top of the **defflavor** form, or be in the list of **:required-instance-variables**.

defflavor offers an alternate syntax enabling you to explicitly specify the keyword to be used to initialize an instance variable. For information on it, see the section "Specifying Names for Functions that Access Instance Variables".

(**:locatable-instance-variables** *vars...*)
Enables you to use **locf** to get a locative pointer to the cell inside an instance that contains the value of an instance variable as follows:

```
(locf (accessor object))
```

If the accessor function is **ship-mass**, and the object is **my-ship**, you can get a locative pointer to the cell inside **my-ship** containing the value of the **mass** instance variable as follows:

```
(locf (ship-mass my-ship))
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

Any instance variables specified by **:locatable-instance-variables** are automatically made readable as well; that is, an accessor function is automatically defined.

defflavor offers an alternate syntax enabling you to explicitly specify the names of accessor functions. For information on it, see the section "Specifying Names for Functions that Access Instance Variables".

(**:method-combination** *symbol*)
Declares the way that methods from different flavors are to be combined. The syntax is:

```
(:method-combination
     generic-function name
     generic-function (name args...)
     ...)
```

If the **:method-combination** option is also supplied to **defgeneric**, that option must agree with the **:method-combination** option given to **defflavor**.

For further details on usage and an example, see the section "Using the **:method-combination** Option".

(**:method-order** *generic-function-names*)
The arguments are names of generic functions that are frequently used or for which speed is important. Their handlers are inserted into the handler hash table first, so that they are found by the first hash probe.

(**:mixture** *specs...*)
Defines a family of related flavors. The organization of this family usual-

ly consists of one *basic flavor* and several *mixin flavors*. The flavors in the family are automatically constructed by mixing various mixins with the basic flavor. When **make-instance** is called, it uses its keyword arguments (or defaults to values in the **:default-init-plist** of the **defflavor** form) to choose which flavor of the family to instantiate.

The basic flavor is the one that includes the **:mixture** option in its **defflavor**. By convention, it is often named **basic-***foo*. Often the basic flavor is not intended to be instantiated. If so, you should supply the **:abstract-flavor** option to specify that.

The names for the family members are chosen automatically. The name of such an automatically constructed flavor is a concatenation of the names of its components, separated by hyphens. Note that obvious redundancies are removed heuristically. An example is shown later in this section.

defflavor of the basic flavor defines the automatically constructed flavors as well as the basic flavor. Similarly, **compile-flavor-methods** of the basic flavor also compiles combined methods of the automatically constructed flavors.

The **:mixture** option is not inherited by flavors from their component flavors. You can still build a new flavor from a flavor that uses **:mixture**, but you should not expect that the new flavor follows the conventions specified by the **:mixture** option of its component. If you try to apply the **:mixture** conventions of a component flavor to the new flavor built on it, you will get a warning. If you want the new flavor to follow the same conventions, you can include the same **:mixture** option in the **defflavor** of the new flavor.

The **:mixture** option has the following form:

```
(:mixture spec spec ...)
```

Each *spec* is processed independently, and all the resulting mixins are mixed together. A *spec* can be any of the following:

(*keyword mixin*)
: Add mixin if the value of *keyword* is **t**; add nothing if **nil**.

(*keyword* (*value mixin*) (*value mixin*) ...)
: Look up the value of *keyword* in this alist and add the specified mixin.

(*keyword mixin subspec subspec* ...)

(*keyword* (*value mixin subspec subspec* ...) ...)

A *subspec* has the same form as a spec. Subspecs are processed only when the specified keyword has the specified value. Use them when there are interdependencies among keywords.

A *mixin* is one of the following:

symbol	The name of a flavor to be mixed in.
nil	No flavor needs to be mixed in if the keyword takes on this value.
string	This value is invalid: Signal an error with the string as part of the message.

A *value* can be anything that is acceptable as a clause key for **case**. This includes symbols, numbers, characters, instances, and named structures; but excludes lists, strings, and arrays other than named structures. The symbol **otherwise** is treated specially, as in **case**. For example, if you want to allow for values of **:size** other than 1, the **:mixture** clause is:

```
(:mixture (:size (1 small-mixin)
                 (otherwise nil)))
```

make-instance checks that the keywords are given with valid values, if you do not have **otherwise** as a clause key. You need an **:init-keywords** declaration for any keywords that are used only in the **:mixture** declaration. You cannot specify constructors for the flavors generated by the **:mixture** option.

The following example defines a basic flavor called **cereal-stream**:

```
(defflavor cereal-stream (...) (stream)
  ...
  :abstract-flavor
  (:init-keywords :characters :direction
                  :ascii :hang-up-when-close)
  (:mixture (:characters
             (t nil (:direction
                       (:in buffered-line-input-stream)
                       (:out buffered-output-character-stream))
                    (:ascii ascii-translating-character-stream))
             (nil nil (:direction (:in buffered-input-stream)
                                  (:out buffered-output-stream))
                      (:ascii "Ascii translation is not
                               meaningful for binary streams")))
            (:hang-up-when-close hang-up-when-close-mixin)))
```

The declaration above indicates that the basic flavor **cereal-stream** cannot be instantiated alone. The **:direction** option and an appropriate value (**:in** or **:out**) must be provided to **make-instance**. The **:characters** option does not itself add any mixins (hence the **nil**), but the processing of the **:direction** option depends on the value of the **:characters** option, which selects a character stream or a binary stream. The **:ascii** option is allowed only for character streams, and an error message is specified if it is used with a binary stream. If **:ascii** had not been mentioned in the **:characters nil** case, the keyword would have been ignored by **make-instance** on the assumption that a **make-instance** method was going to use it. Any kind of **cereal-stream** can have a **:hang-up-when-close** option.

You could make an instance of a member of this family as follows:

```
(make-instance 'cereal-stream :characters t
                              :direction :in
                              :hang-up-when-close t)
```

The name of the flavor that is instantiated is: **hang-up-when-close-buffered-line-input-cereal-stream**.

:no-vanilla-flavor

Normally when a flavor is defined, the special flavor **flavor:vanilla** is included automatically at the end of its list of components. The **flavor:vanilla** flavor provides some default methods for several useful functions that all objects are supposed to understand, including **sys:print-self** and **:which-operations**, among others.

If any component of a flavor specifies the **:no-vanilla-flavor** option, **flavor:vanilla** is not included in that flavor. This option should not be used casually.

(**:ordered-instance-variables** *symbols*)

:ordered-instance-variables increases efficiency at the cost of dynamic modification. This is an advanced option that can be used to micro-optimize certain programs. It should not be used casually. The arguments are names of instance variables that must appear first (and in this order) in all instances of this flavor or any flavor depending on this flavor. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

When you use this option, you must specify all of this flavor's instance variables in the list of instance variables at the top of the **defflavor** form, including any inherited instance variables. When you build a flavor from another flavor that uses **:ordered-instance-variables**, both flavors must specify the same order of instance variables. The new flavor can add instance variables, but they must come after the inherited instance variables. For example:

```
(defflavor latched-object (simple-latch)
           ()
  (:ordered-instance-variables simple-latch))

(defflavor ordered-lock (simple-latch next-lock number)
           (latched-object)
  (:ordered-instance-variables simple-latch next-lock number))
```

If you try to redefine a flavor that has ordered instance variables, you will notice that the actual order of the instance variables does not change once the Flavors system has committed to a particular order. Flavors gives you a warning when this occurs. However, you can change the order by reloading the code into a world in which the flavor has not yet been defined.

(**:readable-instance-variables** *vars...*)

Creates an accessor function for querying an object for the value of each of the specified instance variables. The name of the function is the name of the flavor, followed by a dash "-", followed by the name of the instance variable. The accessor function for the instance variable *v* of flavor *f* is:

f-v

For example, the **ship** flavor has an instance variable named **mass**. If the **:readable-instance-variables** option is given, an accessor function named **ship-mass** is created. You can use it on an instance of **ship** as follows:

```
(ship-mass my-ship)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

This function is created in the package that is current at macro-expansion time. **defflavor** offers an alternate syntax for specifying the full name of accessors explicitly, which enables you to specify the package as well as the function name. See the section "Specifying Names for Functions that Access Instance Variables".

You can use "**:conc-name** Option for **defflavor**" to specify a different name for the accessor function.

(**:required-flavors** *flavor-names*)

Specifies flavors that must be included as components (directly or indirectly) in a new flavor incorporating this one, if the flavor is to be instantiated. The arguments are names of the required flavors. Typically the **:required-flavors** option is used when defining a mixin flavor to specify which base flavor or flavors it requires.

The difference between giving the **:required-flavors** option and listing them directly as components at the top of the **defflavor** form is that the **:required-flavors** option does not make any commitments about where those flavors should appear in the ordered list of components. The order of the **:required-flavors** is determined by the flavor that does specify them as components.

Declaring a flavor as required using the **:required-flavors** option

- Allows instance variables declared by that flavor or its components to be accessed.
- Allows you to use any internal functions defined for the flavors in the **:required-flavors** clause, or for their components. see the section "Defining Functions Internal to Flavors".

- Provides error checking. An attempt to instantiate a flavor that does not include the required flavors as components signals an error.

(**:required-init-keywords** *init-keywords*)

Specifies keywords that must be supplied when making an instance of this flavor. The arguments are the required keywords. It is an error to try to make an instance of this flavor or incorporate it without specifying these keywords as arguments to **make-instance** or as a **:default-init-plist** option in a component flavor. This error is often detected at compile-time.

Each keyword argument to be accepted by **make-instance** must be declared by one of the following options: **:initable-instance-variables**, **:init-keywords**, **:area-keyword**, or be the default area-keyword **:area**. Note that the **:required-init-keywords** option does not imply any one of the above four, so any **:required-init-keywords** must also be declared in one of ways listed above.

(**:required-instance-variables** *vars...*)

Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. The arguments are the required instance variables. It is an error to try to make an instance of this flavor or a flavor incorporating it if it does not have the required instance variables.

Required instance variables can be freely accessed by methods just like normal instance variables. The difference between using the **:required-instance-variables** option and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

(**:required-methods** *generic-function-names*)

Specifies generic functions that must be supported with methods by a new flavor incorporating this one, if the flavor is to be instantiated. The arguments are the names of the generic functions that must be supported with methods.

It is an error to instantiate such a flavor if it lacks a method for one of these generic functions. When the **:required-methods** option is given, it is possible for the error to be detected when the flavor is defined (usually at compile-time), rather than at run-time.

Typically this option appears in the **defflavor** form for a base flavor. Usually this is used when a base flavor calls a generic function on itself, but the base flavor does not provide a method for that generic function. **:required-methods** indicates that the base flavor cannot be instantiated alone, but must be instantiated with other components (mixins) that do handle the required generic functions.

(**:settable-instance-variables** *vars...*)

This option is available for compatibility with the old message-passing flavor system. When writing new code, it is good practice to use the **:writable-instance-variables** option instead.

Enables automatic generation of methods for setting the values of instance variables. The message name is "**:set-**" followed by the name of the variable. All settable instance variables are also automatically made gettable and initable.

To use the automatically-generated method, use the old **send** syntax. For example:

```
(send my-ship :set-mass 100)
```

You can give this option with arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

(**:special-instance-variables** *vars...*)

This option is intended for internal system uses only. It is documented for completeness.

Use the **:special-instance-variables** option if you need instance variables to be bound as special variables to values in the instance, when certain methods are called. (The methods are specified with **:special-instance-variable-binding-methods**.) This option detracts from performance and should be avoided.

The format of **:special-instance-variables** is the same as that of **:readable-instance-variables**. If the option is given alone, all instance variables are bound. If the option is given in the format **(:special-instance-variables a b c)**, only the variables **a**, **b**, and **c** are bound.

(**:special-instance-variables-binding-methods** *generic-function-names*)

This option is intended for internal system uses only. It is documented for completeness.

This option specifies names of generic functions and messages that should cause any instance variables declared **:special-instance-variables** to be bound as special variables to values in the instance.

(**:writable-instance-variables** *vars...*)

Enables you to use **setf** to set the value of an instance variable as follows:

```
(setf (accessor object) value)
```

If the accessor function is **ship-mass**, and the object is **my-ship**, you can set the value of the **mass** instance variable to 100 as follows:

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

You can give this option arguments, to specify the instance variables to which it applies. If no arguments are provided, this option applies to all instance variables listed at the top of the **defflavor** form.

Any instance variables specified by **:writable-instance-variables** are automatically made readable as well.

For information on an alternate syntax of **defflavor** which see the section "Specifying Names for Functions that Access Instance Variables".

Specifying Names for Functions that Access Instance Variables

This section describes an alternate format of **defflavor** that enables you to explicitly specify the name of the generic functions that read, write, or locate an instance variable. Using this format, you specify the name of the reader function; note that the name of the writer function always involves **setf** and the locator function involves **locf**. You can also explicitly specify the keyword used to initialize an instance variable. If you are using message-passing, this flexible syntax also applies to gettable and settable instance variables.

This example exhibits the syntax of all of these:

```
(defflavor test-flavor (a b c d e f g h i j k l m n o p) ()
  (:initable-instance-variables a b (:sea c))
  (:settable-instance-variables d e (:change-f f))
  (:gettable-instance-variables g h (:eye i))
  (:readable-instance-variables j (get-kay k))
  (:writable-instance-variables l (m-value m))
  (:locatable-instance-variables n (get-o o)))
```

When making an instance of **test-flavor**, you can initialize the variables **a**, **b**, and **c** as follows:

```
(setq test-instance (make-instance 'test-flavor :a 2
                                                :b 4
                                                :sea 6))
```

You can set the value of the variables **d**, **e** and **f** as follows:

```
(send test-instance :set-d 22)
(send test-instance :set-e 44)
(send test-instance :change-f 66)
```

You can get the value of the variables **g**, **h** and **i** as follows:

```
(send test-instance :g)
(send test-instance :h)
(send test-instance :eye)
```

You can read the values of the variables **j**, **k**, **l**, **m**, **n** and **o** as follows:

```
(test-flavor-j test-instance)
(get-kay test-instance)
(test-flavor-l test-instance)
(m-value test-instance)
(test-flavor-n test-instance)
(get-o test-instance)
```

You can write the value of the variable **m** as follows:

```
(setf (m-value test-instance) 33)
```

You can locate the value of the variables **n** and **o** as follows:

```
(locf (test-flavor-n test-instance))
(locf (get-o test-instance))
```

Specifying defflavor Options More than Once

Some **defflavor** options may be specified more than once. Those that are not allowed to be specified more than once cause a warning to occur if you do so. This is the list of options that can be specified more than once:

:constructor
:default-init-plist
:functions
:gettable-instance-variables
:init-keywords
:initable-instance-variables
:locatable-instance-variables
:method-combination
:method-order
:ordered-instance-variables
:readable-instance-variables
:required-flavors
:required-init-keywords
:required-instance-variables
:required-methods
:settable-instance-variables
:special-instance-variables
:special-instance-variable-binding-methods
:writable-instance-variables

Advanced Concepts for defmethod

defmethod Declarations

It is legal to give the same **declare** statements to **defmethod** as are accepted by **defun**. **defmethod** also accepts these additional **declare** statements:

(declare (flavor:solitary-method))

If this declaration is used, and only one method is available for the generic function, Flavors implements the generic function with the goal of saving space, and at the expense of making the method slower to call. The generic function calls the method directly, bypassing the usual dispatching mechanism.

If more than one method is available for the generic function, this declaration has no effect. Also, this declaration saves space only when a method is inherited by more than one flavor.

The input editor illustrates the use of solitary methods. Each input editor command is a method. There are many commands, and there is no need for them to be any faster than the user can type them. Declaring these commands solitary methods optimizes space.

(declare (flavor:inhibit-arglist-checking))

Normally, Flavors checks the arguments of **defmethod** forms to ensure that they are consistent with the arguments expected by the generic function. This declaration prevents that consistency check from occurring. For example:

```
(defmethod (accept encapsulating-output-stream)
           (presentation-type &rest args)
  (declare (flavor:inhibit-arglist-checking))
  (lexpr-send stream 'accept presentation-type :stream
              self args))
```

Implicit Blocks for Methods

The interpreter and compiler generate implicit blocks for functions whose name is a list (such as methods) just as they do for functions whose name is a symbol. You can use **return-from** for methods. The name of a method's implicit block is the name of the generic function it implements. If the name of the generic function is a list, the block name is the second symbol in that list.

Variant Syntax of defmethod

The following variant **defmethod** syntax is supported, but rarely used:

(defmethod (*generic-function flavor options..*) *symbol*)

symbol is the name of an ordinary function (not a generic function) that is to be used as the method for performing *generic-function* on instances of the given *flavor*.

For example:

```
(defmethod (delete-file logical-pathname) logical-pathname-pass-on)
```

symbol cannot be an internal flavor function defined by **defun-in-flavor**, **defmacro-in-flavor**, or **defsubst-in-flavor**. You can use the body of the normal **defmethod** syntax to call an internal flavor function to perform the operation.

Function Specs for Flavor Functions

This section tells what the function specs are for various types of flavor functions, such as generic functions, methods, and wrappers. For more detailed information on how to use function specs: See the section "Function Specs".

Type of Function	*Function Spec*
Combined method	**(flavor:combined** *generic-function flavor***)**
Generic function	Its name, usually a symbol
Handler	**(:handler** *generic-function flavor***)**
Internal function	**(defun-in-flavor** *function-name flavor***)**
Internal macro	**(defun-in-flavor** *macro-name flavor***)**
Internal substitutable function	**(defun-in-flavor** *subst-name flavor***)**
Locator function	**(locf** *function***)**
Method	**(flavor:method** *generic-function flavor options..***)**
Setter function	**(setf** *generic-function***)**
Whopper	**(flavor:ncwhopper** *generic-function flavor***)**
Wrapper	**(flavor:wrapper** *generic-function flavor***)**

A note to Zetalisp programmers: the function spec of a setter function is **(setf** *generic-function***)**, not **(zl:setf** *generic-function***)**.

Setter and Locator Function Specs

A setter function is a generic function that sets an instance variable. A locator function is a generic function that locates an instance variable.

You invoke a setter function by typing something like:

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

The function spec for the above setter function is **(setf ship-mass)**.

Similarly, you invoke a locator function by typing:

```
(locf (ship-mass my-ship))
```

The function spec for the above locator function is **(locf ship-mass)**.

A setter function can be defined in the following ways:

- Automatically: by providing the **:writable-instance-variables** option for **defflavor**. For example:

```
(defflavor ship (x-velocity y-velocity mass)
           ()
  :writable-instance-variables)
```

- Implicitly: by using **defmethod** to define a method, which in turn creates a generic function. For example:

```
(defmethod ((setf ship-mass) ship) (new-mass)
  (setq mass new-mass
        mass-squared (* new-mass new-mass))
  new-mass)
```

- Explicitly: by defining the generic function **(setf ship-mass)** using **defgeneric**. This would be unusual.

It is not necessary to use **defsetf** to tell **setf** how to deal with setter functions such as **ship-mass**. The existence of the function named **(setf ship-mass)** is enough for **setf** to know what to do.

Note that not all setter functions have the function spec **(setf *function*)**. Only those that were defined in one of the three ways described above have such a function spec. Setter functions defined with **defsetf** do not have such a function spec. Here is an example of using the setter function spec:

```
(let ((setter (if cond #'(setf foo) #'(setf bar))))
  (dolist (thing things)
    (funcall setter thing nil)))
```

Property List Methods

It is often useful to associate a property list with an abstract object, for the same reasons that it is useful to have a property list associated with a symbol. This section describes a mixin flavor that can be used as a component of any new flavor in order to provide that new flavor with a property list.

sys:property-list-mixin *Flavor*

Provides methods that perform the generic functions on property lists. **sys:property-list-mixin** provides methods for the following generic functions:

:get *indicator* *Message*

Looks up the object's *indicator* property. If it finds such a property, it returns the value; otherwise it returns **nil**.

:getl *indicator-list* *Message*

Like the **:get** message, except that the argument is a list of indicators. The **:getl** message searches down the property list for any of the indicators in *indicator-list* until it finds a property whose indicator is one of those elements. It returns the portion of the property list beginning with the first such property that it found. If it does not find any, it returns **nil**.

:putprop *property indicator* *Message*

Gives the object an *indicator*-property of *property*.

:remprop *indicator* *Message*

Removes the object's *indicator* property by splicing it out of the property list. It returns that portion of the list inside the object of which the former *indicator*-property was the **car**.

:push-property *value indicator* *Message*

The *indicator*-property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This message sets the *indicator*-property of the object to a list whose **car** is *value* and whose **cdr** is the former *indicator*-property of the list. Executing the form

```
(send object :push-property value indicator)
```

is analogous to doing

```
(zl:push value (send object :get indicator))
```

See the function **zl:push**.

:property-list *Message*

Returns the list of alternating indicators and values that implements the property list.

:set-property-list *list* *Message*

Sets the list of alternating indicators and values that implements the property list to *list*.

(flavor:method :property-list sys:property-list-mixin) *list* *Init Option*

Initializes the list of alternating indicators and values that implements the property list to *list*.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

Generic Functions and Messages Supported by flavor:vanilla

This section lists the generic functions and messages for which **flavor:vanilla** provides a method.

Any flavor can override a **flavor:vanilla** method by providing a method for one of the operations listed below.

You can write a method for **flavor:vanilla**. No method defined for **flavor:vanilla** can access instance variables, because the flavor has none. Note that when you write a method for **flavor:vanilla**, it takes a long time for the Flavor system to update all flavors dependent on **flavor:vanilla**.

flavor:vanilla *Flavor*

This flavor is included in all flavors by default. **flavor:vanilla** has no instance variables, but it provides several basic useful methods, some of which are used by the Flavor tools.

Every flavor has **flavor:vanilla** as a component flavor, unless you specify not to include **flavor:vanilla** by providing the **:no-vanilla-flavor** option to **defflavor**. It is unusual to exclude **flavor:vanilla**.

For a summary of all functions, macros, special forms, and variables related to Flavors: See the section "Summary of Flavor Functions and Variables".

flavor:vanilla provides methods for the following generic functions:

:describe Calls **flavor:describe-instance**, which prints the following information onto the ***standard-output*** stream: a description of the instance, the name of its flavor, and the names and values of its instance variables. It returns the instance.

get-handler-for Returns the given object's method for a specified operation, or **nil** if the object has no method for the operation.

:operation-handled-p Returns **t** if the given object has a handler for the specified operation, **nil** if it does not.

sys:print-self Produces a printed representation of the given object.

:send-if-handles Performs the specified generic function if the given object has a method defined for it; otherwise returns **nil**.

:which-operations Returns a list of generic functions and messages that the given object supports with methods.

Copying Instances

Flavors does not include any built-in way to copy instances. In order to copy an instance you must understand a lot about the instance. You must understand the semantics of the instance variables, so that their values can be copied if necessary. You must understand the instance's relations to the external environment so that new relations can be established for the new instance. At a more basic level, you must understand whether it makes sense to copy this particular instance at all.

Copying is conceptually a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to the flavor of that instance. Modularity dictates that this knowledge be contained in the instance's flavor, not in a general copying function. Thus the way to copy an instance is to use a method that takes care of all the ramifications of making a copy.

Copying instances raises a number of issues, such as:

- Should any **make-instance** methods be applied to the new instance? If so, what arguments should be supplied to those methods?

- If the instance has a property list, you should copy the property list (for example, with **copy-list**) so that sending a **:putprop** or **:remprop** message to one of the instances does not affect the properties of the other instance.

- The instance might be contained in data structure maintained by the program of which it is a part. For example, a graphics system might have a list of all the objects that are currently visible on the screen. Copying such an instance requires making the appropriate entries in the data structure. Some of the following points are specific examples of this general point.

- If the instance is a pathname, the concept of copying is not even meaningful. Pathnames are *interned*, which means that there can only be one pathname object with any given set of instance-variable values.

- If the instance is a stream connected to a network, some of the instance variables represent an agent in another host elsewhere in the network. Copying the instance requires that a copy of that agent somehow be constructed.

- If the instance is a stream connected to a file, should copying the stream make a copy of the file or should it make another stream open to the same file? Should the choice depend on whether the file is open for input or for output?

Because copying instances involves so many semantic issues, Flavors does not provide a default method for copying an instance, nor does it suggest a standard name for the generic function that does the copying.

One way that programmers have organized copying of instances is to define a generic function that uses **:append** method combination. Each component flavor controls the copying of its own aspect of the instance's behavior, by supplying some **make-instance** arguments. Inside the generic function, **make-instance** is

called with the appended list as its arguments. Each component flavor has a **make-instance** method that extracts the keyword arguments that are relevant to it and initializes the appropriate aspect of the new instance. A simple example follows:

```
(defflavor copyable-object () ())

(defgeneric make-copy (copyable-object)
  (:method-combination :append :most-specific-first)
  ;; call make-instance with the correct arguments,
  ;; which come from the values of each method
  (:function
    (apply
      #'make-instance
      (type-of copyable-object)
      (funcall (flavor:generic make-copy) copyable-object))))

(defflavor copyable-property-list-mixin () (si:property-list-mixin)
  (:init-keywords :properties))

(defmethod (make-copy copyable-property-list-mixin) ()
  ;;ensure that the properties get copied
  `(:properties ,(copy-list (send self :property-list))))

(defmethod (make-instance copyable-property-list-mixin)
           (&key properties &allow-other-keys)
  (send self :set-property-list properties))

(defflavor color-mixin (color) ()
  :initable-instance-variables
  (:readable-instance-variables (get-color color)))

(defmethod (make-copy color-mixin) ()
  ;;ensure that the color instance variable gets copied
  `(:color ,color))

(defflavor example () (copyable-property-list-mixin
                       color-mixin
                       copyable-object))

(setq *obj* (make-instance 'example :color :red
                           :properties '(value 1 other-value 2)))

(get-color *obj*) => :RED
(send *obj* :get 'value) => 1

(setq *new-obj* (make-copy *obj*))
```

```
;;;the new object has the same properties as the old
(get-color *new-obj*) => :RED
(send *new-obj* :get 'value) => 1

;;;the new object has its own copy of the property list
(send *new-obj* :putprop 5 'value) => 5
(send *new-obj* :get 'value) => 5
(send *obj* :get 'value) => 1
```

A related feature is the **:fasd-form** message, which provides a way for an instance to tell the compiler how to copy it from one Lisp world into another, via a bin file. This is different than making a second copy of the instance in the same Lisp world. **:fasd-form** is a way to get an *equivalent* instance when the bin file is loaded.

Note on make-instance and :fasd-form

Flavor instances are dumped as the forms which are evaluated to create them. As long as the creating forms do not change, flavor instances are compatible across releases and machine architectures. It is always possible that the syntax of **make-instance** will change from one release to another. For this reason, we suggest that you avoid returning **make-instance** from **:fasd-form**, but instead define your own function with a name such as **make-foo-for-loading-from-file** and have **:fasd-form** return a call to that function.

If you later need to change something, you can make your **:fasd-form** start returning calls to a second function such as **make-foo-for-loading-from-file-version-2**. You can keep the previous function **make-foo-for-loading-from-file** around for compatibility with old files, so you can continue to load them.

The Order of Defining Flavors and Flavor Operations

Programmers have a certain amount of freedom in the order in which they do **defflavor**s, **defmethod**s, and **defwrapper**s. This freedom makes it easy to load programs containing complex flavor structures without having to do things in a certain order. Flavors does not require that all methods and operations for a flavor must be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

- Before any flavor operation can be defined, the flavor must have been defined with **defflavor**. This rule applies for **defmethod**, **defwhopper**, **defwhopper-subst**, **defwrapper**, **defun-in-flavor**, **defmacro-in-flavor**, and **defsubst-in-flavor**. This is necessary because the system needs a place to remember the method (or other type of operation), and must know the flavor's instance variables in order to compile the method.

- A flavor can be defined before its component flavors have been defined. This allows **defflavor**s to be spread between files according to a program's modularity.

- In some cases, a method can be defined for a flavor before all of the component flavors are defined. However, if the method accesses an instance variable for a component flavor that is not yet defined, the method will not work. In these cases, compiling those methods produces a warning that an instance variable was declared special (because the system did not realize it was an instance variable).

- Any methods automatically generated by the **defflavor** options **:readable-instance-variables** and **:writable-instance-variables** are generated at the time the **defflavor** is done.

- At the time that a flavor is "composed", it is necessary that all of its component flavors must be defined. A flavor is composed when **compile-flavor-methods** is evaluated, or when the first instance is made. When a flavor is composed, the system performs some error-checking (such as the check that **:required-instance-variables** are included); generates the combined methods for the flavor; and generates any constructor functions indicated by the **:constructor** option for **defflavor**. Note that **compile-flavor-methods** generates combined methods and constructors at compile-time, and verifies that they are still correct at load-time. If any of the combined methods or constructors have become obsolete (for example, if the definition of the flavor has changed), they are regenerated at load-time.

- Because internal functions of flavors are lexically scoped, they must be declared or defined before their callers. Otherwise the caller would not be compiled or evaluated in the proper lexical environment, and would not know that the name of the internal function refers to an internal function rather than to an ordinary function in the global environment. The **:functions** option to **defflavor** can be used to declare the names of **defun-in-flavor** internal functions of the flavor. See the section "Defining Functions Internal to Flavors".

Implementation of Flavors

This section describes the Symbolics implementation of Flavors. There is no need to read or understand this section to use Flavors.

Efficiency Considerations of Flavors

The efficiency philosophy of Flavors is to optimize run-time speed to the maximum extent that does not compromise other goals, such as the flexibility to redefine anything while the program is running. In addition to Flavors-related goals, general Symbolics system goals, such as full run-time error checking, avoiding widespread use of declarations, and providing the best functionality, are not compromised for the sake of efficiency.

CPU time and page fault rate determine response time, so they are more important than virtual memory size, which only consumes inexpensive disk storage. Consequently Flavors maintains multiple copies of information when that improves virtual memory locality or execution speed. Keeping those multiple copies consistent slows down program development operations, especially when modifying flavors that have hundreds of dependents. This tradeoff is acceptable, since development operations need not be faster than human speeds (several seconds), while run-time operations must operate at computer speeds (microseconds). Speeding up the run-time operations also speeds up the development tools built on them.

The key areas that are important to optimize in an object-oriented programming system are:

- Selection of one or more methods when a generic function is called
- Instance variable access from a method
- Instance creation

The following sections discuss how Flavors implements these operations.

Implementation of Method Selection

The first time a flavor is instantiated, or during compilation of the program if so directed, a handler function is precomputed for each generic function that the flavor supports. The method-combination procedure selects a set of methods and produces Lisp code that combines them. If the code can be optimized into calling a single method, that method is the handler. Otherwise a combined method is generated, compiled to machine code, and used as the handler. The combined method calls the methods with ordinary Lisp function calls.

The results of this precomputation are saved in a *handler table* associated with the flavor, keyed by the generic function. Thus when a generic function is called, the method selection process consists of finding the instance's flavor, looking in the handler table, and calling the handler. The handler table is a hash table whose structure is optimized to exploit the pipelined characteristics of the 3600's memory bus.

Subsequent changes to the program such as adding methods, removing methods, declaring a different method combination type, or changing a flavor's components incrementally update all affected handler tables, compiling new combined methods when necessary. For this purpose each flavor is linked to the flavors that depend on it and each combined method records how it was generated.

Implementation of Instance Variable Access

An instance is represented as a block of storage whose first word references the flavor and whose remaining words contain values of instance variables. Offsets within the instance of instance variables cannot be compiled into methods that access the variables. These offsets are variable at run time, depending on the flavor of the instance, which can be any flavor that has the method's flavor as a component. Multiple inheritance makes it impossible to allocate fixed offsets to instance variables, because two flavors using the same offset for different instance variables might later be mixed together and one of them would have to change.

The solution is to use indirect addressing. Each entry in a handler table includes a *mapping table*, which contains instance variable offsets. A method receives a mapping table as an argument. Offsets into the mapping table are compiled into methods. These offsets don't change, because when two flavors are mixed together each has its own mapping table. Accessing an instance variable fetches the instance variable's offset from the mapping table, adds it to the address of the instance, and references that memory location.

When a combined method calls another method, it supplies a mapping table fetched from its own mapping table. Thus mapping tables actually form a tree structure parallel to the tree structure of flavor components.

In principle every flavor needs a separate mapping table for each component, and the total number of mapping tables could be proportional to the square of the number of flavors. In practice the average number of components of a flavor is small and only components that have instance variables need mapping tables. Thus the average number of mapping tables per flavor is only 4.1 and the total memory occupied by mapping tables is negligible.

Flexible Representation of Instances

Redefining a flavor in a way that changes the representation of instances, such as adding or deleting an instance variable, arranges for existing instances to be updated automatically. It makes a new flavor (with the same name) and changes the old flavor's handler table so that all generic functions rearrange the instance, change its flavor reference to the new flavor, and retry the operation. If the new instance representation is larger, rearranging the instance allocates new storage, copies the instance variable values into it, and deposits forwarding addresses with a **sys:dtp-structure-forward** and **sys:dtp-element-forward** tag into the old storage. The instance variable accessing instructions and the garbage collector recognize these forwarding pointers.

Implementation of Instance Creation

The first time a flavor is instantiated, the initialization information from all its components is combined and saved in a convenient form. Subsequent instantiations consist of allocating storage, copying a template instance, initializing any instance variables whose initial values are not constant, and invoking initialization methods if any have been defined.

Using Message-Passing Instead of Generic Functions

Message-passing is supported for compatibility with previous versions of the flavor system. This section describes the features that support message-passing. When writing new programs, it is good practice to use generic functions instead of message-passing.

Defining Methods to Be Called by Message-Passing

The syntax for **defmethod** is as follows:

(defmethod *(generic-function flavor options..) (arg1 arg2..) body..*)

To define a method to be invoked by sending a message (instead of calling a generic function), supply a keyword as the *generic-function* argument to **defmethod**. The keyword is the name of the message. For example:

```
;;; define a message :speed-of, to be used with send syntax
;;; on instances of the ship flavor

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

This method should be invoked by the old **send** syntax:

(send *instance message args...*)

For example:

```
(send my-ship :speed-of)
```

You can also specify that any methods for a certain flavor should be invocable both by generic functions and messages. To do so, supply the **:compatible-message** option to **defgeneric**. Thus, any methods for that generic function can be called with the generic function syntax, or the old **send** syntax.

For any methods invocable by messages, you can call **defgeneric** to update the flavor to treat those methods as generic functions. If you do so, the old **send** syntax no longer works.

Defining a Compatible Message for a Generic Function

The **:compatible-message** option to **defgeneric** indicates that any methods written for this generic function should be callable in two ways: by calling the generic function or by sending a message. One example of the use of **:compatible-message** is in conjunction with a programming construct that recognizes messages only as selectors, such as **defselect**.

The name of the generic function is given as an argument to **defgeneric**. The name of the message is given as an argument to the **:compatible-message** option.

For example:

```
(defgeneric speed (moving-objects)
   (:compatible-message :speed-of))

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

You can invoke the generic function **speed** as follows:

```
(speed my-ship)
```

You can invoke the same method by sending the **:speed-of** message as follows:

```
(send my-ship :speed-of)
```

You can also use either **speed** or **:speed-of** in **defmethod**; **:speed-of** is automatically changed to **speed** in the method's function spec.

If you are converting message-passing programs to generic functions, and using **:compatible-message** to define messages so that old programs that use those messages will continue to work, you should be sure that every use of the message you are defining is within your program. If other programs are using the same message name for different purposes, your **defgeneric** form will have an effect on the other methods. For example, all methods for that message are constrained to take the arguments that your generic function takes. Also, when any methods for that message are compiled, they are defined as being methods for your generic function.

Note that if a generic function uses both the **:compatible-message** and **:function** options, the way to trigger the generic dispatch within the **:function** option is by sending the message, not by calling the generic function.

Functions for Passing Messages

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.

send-if-handles *object message* &rest *arguments*
: Sends a message to a flavor instance, if the flavor has a method defined for this message.

lexpr-send-if-handles *object message argument* &rest *arguments*
: Like **send-if-handles**, except that the last element of arguments should be a list. All elements of that list are passed as arguments.

Note that **send-if-handles**, **:send-if-handles** and **lexpr-send-if-handles** work by sending the **:send-if-handles** message. You can customize the behavior of these operations by defining a method for that message.

Conditions

Introduction to Signalling and Handling Conditions

This chapter is tailored for applications programmers. It contains descriptions of all conditions that are signalled by Genera. With this information, you can write your own handlers for events detected by the system, or define and handle classes of events appropriate for your own application.

The chapter covers the following major topics:

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

Overview of Signalling and Handling

An *event* is something that happens during execution of a program that the system can detect, like the effect of dividing by zero. Some events are errors — which means something happened that was not part of the contract of a given function — and some are not. In either case, a program can report that the event has occurred, and it can find and execute user-supplied code as a result.

The reporting process is called *signalling*, and subsequent processing is called *handling*. A *handler* is a piece of user-supplied code that assumes control when it is invoked as a result of signalling. Genera includes default mechanisms to handle a standard set of events automatically.

The mechanism for reporting the occurrence of an event relies on flavors. Each standard class of events has a corresponding flavor called a *condition*. For example, occurrences of the event "dividing by zero" correspond to the condition **sys:divide-by-zero**.

The mechanism for reporting the occurrence of an event is called *signalling a condition*. The signalling mechanism creates a *condition object* of the flavor appropriate for the event. The condition object is an instance of that flavor. The instance contains information about the event, such as a textual message to report, and various parameters of the condition. For example, when a program divides a number by zero, the signalling mechanism creates an instance of the flavor **sys:divide-by-zero**.

Handlers are pieces of user or system code that are bound for a particular condition or set of conditions. When an event occurs, the signalling mechanism searches all of the currently bound handlers to find the one that corresponds to the condition. The handler can then access the instance variables of the condition object to learn more about the condition and hence about the event.

Handlers have dynamic scope, so that the handler that is invoked for a condition is the one that was bound most recently.

The condition system provides flexible mechanisms for determining what to do after a handler runs. The handler can try to *proceed*, which means that the program might be able to continue execution past the point at which the condition was signalled, possibly after correcting the error. Any program can designate *restart* points. This facility allows a user to retry an operation from some earlier point in a program.

Some conditions are very specific to a particular set of error circumstances and others are more general. For example, **fs:delete-failure** is a specialization of **fs:file-operation-failure** which is in turn a specialization of **fs:file-error**. You choose the level of condition that is appropriate to handle according to the needs of the particular application. Thus, a handler can correspond to a single condition or to a predefined class of conditions. This capability is provided by the flavor inheritance mechanism.

How Applications Programs Treat Conditions

This section provides an overview of how applications programs treat conditions.

- A program signals a condition when it wants to report an occurrence of an event.
- A program binds a handler when it wants to gain control when an event occurs.

When the system or a user function detects an error, it signals an appropriate condition, and some handler bound for that condition then deals with it.

Conditions are flavors. Each condition is named by a symbol that is the name of a flavor, for example, **sys:unbound-variable**, **sys:divide-by-zero**, **fs:file-not-found**. As part of signalling a condition, the program creates a condition object of the appropriate flavor. The condition object contains information about the event, such as a textual message to report and various parameters. For example, a condition object of flavor **fs:file-not-found** contains the pathname that the file system failed to find.

Handlers are bound with dynamic scope, so the most recently bound handler for the condition is invoked. When an event occurs, the signalling mechanism searches all of the current handlers, starting with the innermost handler, for one that can handle the condition that has been signalled. When an appropriate handler is found, it can access the condition object to learn more about the error.

Example of a Handler

condition-case is a simple form for binding a handler. For example:

```
(condition-case ()
    (/ a b)
  (sys:divide-by-zero *infinity*))
```

This form does two things:

- Evaluates `(/ A b)` and returns the result.
- Binds a handler for the **sys:divide-by-zero** condition, which applies during the evaluation of `(/ A b)`.

In this example, it is a simple handler that just returns a value. If division by zero

happened in the course of evaluating `(/ A b)`, the form would return the value of ***infinity*** instead. If any other error occurred, it would be handled by the system's default handler for that condition or by some user handler of higher scope.

You can also bind a handler for a predefined class of conditions. For example, the symbol **fs:file-operation-failure** refers to the set of all error conditions in file system operations, such as "file not found" or "directory not found" or "link to nonexistent file", but not to such errors as "network connection closed" or "invalid arguments to **open**", which belong to different classes.

Signalling

You can signal a condition by calling either **signal** or **error**. **signal** is the most general signalling function; it can signal any condition. It allows either a handler or the user to proceed from the error. **error** is a more restrictive version, which accepts only error conditions and does not allow proceeding. **error** is guaranteed never to return to its caller.

Both **signal** and **error** have the same calling sequence. The first argument is a symbol that names a condition; the rest are keyword arguments that let you provide extra information about the error. See the section "Signalling Conditions". Full details on using the signalling mechanism are in that section.

Applications programs rarely need to signal system conditions although they can. Usually, programs that want to signal conditions define their own condition flavor to signal.

Zetalisp Note: Two simpler signalling functions, **zl:ferror** and **zl:fsignal**, are applicable when you want to signal without defining a new condition. These two functions are now obsolete, however; in new programs, use **error** instead of **zl:ferror**, and **cerror** instead of **zl:fsignal**.

It is very important to understand that signalling a condition is not just the same thing as throwing to a tag. **throw** is a simple control-structure mechanism that allows control to escape from an inner form to an outer form. Signalling is a convention for finding and executing a piece of user-supplied code when one of a class of events occurs. A condition handler might, in fact, do a **throw**, but it is under no obligation to do so. User programs can continue to use **throw**; it is simply a different capability with a different application. For more information: See the section "Flow of Control".

Condition Flavors

Symbols for conditions are the names of flavors; sets of conditions are defined by the flavor inheritance mechanism. For example, the flavor **lmfs:lmfs-file-not-found** is built on the flavor **fs:file-not-found**, which is built on **fs:file-operation-failure**, which, in turn, is built on the flavor **error**.

The flavor inheritance mechanism controls which handler is invoked. For example, when a Genera file system operation fails to find a file, it could signal **lmfs:lmfs-file-not-found**. The signalling mechanism invokes the first appropriate handler it

finds, in this case, a handler for **fs:file-not-found**, one for **fs:file-operation-failure**, or one for **error**. In general, if a handler is bound for flavor **a**, and a condition object **c** of flavor **b** is signalled, then the handler is invoked if **(typep c 'a)** is true; that is, if **a** is one of the condition flavors that **b** is built on.

The symbol **condition** refers to all conditions, including simple, error, and debugger conditions. The symbol **error** refers to the set of all error conditions. Figure ! shows an overview of the flavor hierarchy.

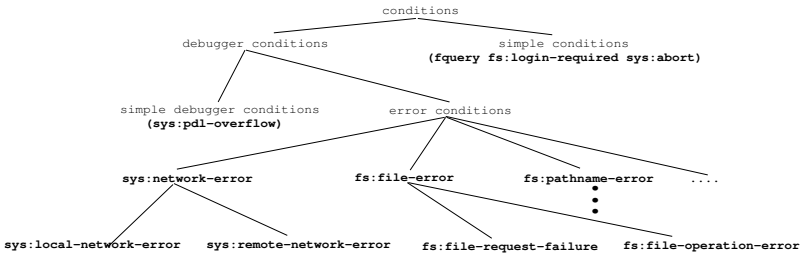


Figure 22. Condition Flavor Hierarchy

error is a base flavor for many conditions, but not all. *Simple conditions* are built on **condition**; *debugger conditions* are built on **dbg:debugger-condition**. *Error conditions* or *errors* are built on **error**. For your own condition definitions, whether you decide to treat something as an error or as a simple condition is up to the semantics of the application.

From a more technical viewpoint, the distinction between simple conditions and debugger conditions hinges on what action occurs when the program does not provide its own handler. For a debugger condition, the system invokes the Debugger; for a simple condition, **signal** simply returns **nil** to the caller.

A warning mechanism also exists; the function **warn** is like the function **signal**, allowing either a handler or the user to proceed from the error. If the variable ***break-on-warnings*** has a **nil** value, **warn** prints its message without entering the Debugger; if ***break-on-warnings*** has a non-**nil** value, **warn** prints its warning message from the Debugger.

Creating New Conditions

An application might need to detect and signal events specific to the application. To support this, you need to define new conditions.

Defining a new condition is straightforward. For simple cases, you need only two forms: One defines the flavor, and the other defines a **dbg:report** method. You can build the flavor definition on either **error** or **condition**, depending on whether or not the condition you are defining represents an error. The following example defines an error condition.

```
(defflavor block-wrong-color () (error))

(defmethod (dbg:report block-wrong-color) (stream)
   (format stream "The block was of the wrong color."))
```

Your program can now signal the error as follows:

```
(error 'block-wrong-color)
```

dbg:report requires one argument, which is a stream for it to use in printing an error message. Its message should be a sentence, ending with a period and with no leading or trailing newline characters.

The **dbg:report** method must not depend on the dynamic environment in which it is invoked. That is, it should not do any free references to special variables. It should use only its own instance variables. This is because the condition object might receive a **dbg:report** message in a dynamic environment that is different from the one in which it was created. This situation is common with **condition-case**.

The above example is adequate, but does not take advantage of the power of the condition system. For example, the error message tells you only the class of event detected, not anything about this specific event. You can use instance variables to make condition objects unique to a particular event. For example, add instance variables **block** and **color** to the flavor so that **error** can use them to build the condition object:

```
(defflavor block-wrong-color (block color) (error)
  :initable-instance-variables
  :readable-instance-variables)

(defmethod (dbg:report block-wrong-color) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

The **:initable-instance-variables** option defines **:block** and **:color** init options; the **:readable-instance-variables** option defines methods for the **block-wrong-color** flavor, which handlers can call.

Your program would now signal the error as follows:

```
(error 'block-wrong-color :block the-bad-block
                          :color the-bad-color)
```

It is a good idea to use **compile-flavor-methods** for any condition whose instantiation is considered likely, to avoid the need for run-time combination and compilation of the flavor. See the macro **compile-flavor-methods**. Otherwise, the flavor must be combined and compiled the first time the event occurs, which causes perceptible delay.

The only other interesting thing to do when creating a condition is to define proceed types. See the section "Proceeding".

Creating a Set of Condition Flavors

You can define your own sets of conditions and condition hierarchies. Just create a new flavor and build the flavors on each other accordingly. The base flavor for the set does not need a **dbg:report** method if it is never going to be signalled itself. For example:

```
(defflavor block-world-error () (error))

(defflavor block-wrong-color (block color) (block-world-error)
  :initable-instance-variables)

(defflavor block-too-big (block container) (block-world-error)
  :initable-instance-variables)

(defmethod (dbg:report block-too-big) (stream)
  (format stream "The block ~S is too big to fit in the ~S."
                 block container))

(defmethod (dbg:report block-wrong-color) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

Establishing Handlers

What is a Handler?

A handler consists of user-supplied code that is invoked when an appropriate condition signal occurs. Genera includes default handlers for all standard conditions. Application programs need not handle all conditions explicitly, but can provide handlers for just the conditions most relevant to the needs of the application.

Classes of Handlers

There are five classes of handlers:

- Bound
- Default
- Global
- Interactive
- Restart

Bound Handlers

The simplest form of handler handles every error condition, each in the same way. The form for binding this handler is **ignore-errors**. In addition, three basic forms

are available to bind handlers for particular conditions. Each of these has a standard version and a conditional variant. In the conditional variants, the handlers are bound only if some expression is true.

Basic Forms for Bound Handlers

condition-bind *list* &body *body*
: This is the most general form. It allows the handler to run in the dynamic environment in which the error was signalled and to try to proceed from the error.

condition-bind-if *cond-form list* &body *body*
: The conditional variant of **condition-bind**.

condition-case *(*&rest *varlist) form* &rest *clauses*
: This is the simplest form to use. It returns to the dynamic environment in which the handler was bound and so, does not allow proceeding.

condition-case-if *(cond-form* &rest *varlist) form* &rest *clauses*
: The conditional variant of **condition-case**.

condition-call *(*&rest *varlist) form* &body *clauses*
: This is a more general version of **condition-case**. It uses user-specified predicates to select the clause to be executed.

condition-call-if *(cond-form* &rest *varlist) form* &body *clauses*
: The conditional variant of **condition-call**.

ignore-errors &body *body*
: Sets up a very simple handler on the bound handlers list that handles all error conditions. If an error signal occurs while *body* is executing, **ignore-errors** immediately returns, with **nil** as its first value and something not **nil** as its second value.

Default Handlers

Default handlers are examined by the signalling mechanism only after all of the bound handlers have been examined. Thus, handlers established by a **condition-bind** that is dynamically outside of a **condition-bind-default** can take precedence over handlers established by **condition-bind-default**. In all other respects, the binding forms work like those for bound handlers.

For example, consider this program:

```
(condition-bind ((a a-handler))
    (condition-bind-default ((b b-handler))
        ...))
```

In this example, `a-handler` is established by a **condition-bind** that is dynamically outside of a **condition-bind-default**. In this case, the **condition-bind** handler, `a-handler`, takes precedence over the **condition-bind-default** handler, `b-handler`.

Bound handlers always take precedence over default handlers, *even* if the default handler is on the inside of the bound handler. Regardless of how they are nested, the whole list of bound handlers is searched before the list of default handlers.

Basic Forms for Default Handlers

condition-bind-default *list* &body *body*
: This is a variant of **condition-bind**. It binds a handler on the default condition list instead of the bound condition list. The distinction is described in these two sections. See the section "Signalling Conditions". See the section "Default Handlers and Complex Modularity".

condition-bind-default-if *cond-form list* &body *body*
: The conditional variant of **condition-bind-default**.

Global Handlers

A global handler is like a bound handler, with an important exception: Unlike a bound handler, which is of dynamic extent, a global handler is explicitly defined and is of *indefinite* extent. (Whereas **condition-bind** is like *binding* a special variable, **define-global-handler** is like *setting* a special variable.) Once defined, a global handler must, therefore, be specifically removed.

Caution: The global handler functions do not maintain the order of the global handler list in any way. If the conditions of two handlers overlap each other in such a way that either handler could handle some instantiable condition, then either handler might run, depending on the order in which they were defined.

Basic Forms for Global Handlers

define-global-handler *name condition arglist* &body *body*
: Defines a global handler function named *name*.

undefine-global-handler *name*
: Removes the global handler function named *name*.

dbg:describe-global-handlers
: Displays the list of conditions for which global handlers have been defined, as well as a list of these handlers.

Application: Handlers Examining the Stack

condition-bind handlers are invoked in the dynamic environment in which the error is signalled. Thus, the Lisp stack still holds the frames that existed when the error was signalled. A handler can examine the stack using the functions described in this section.

One important application of this facility is for writing error logging code. For example, network servers might need to keep providing service even though no user is available to run the Debugger. By using these functions, the server can record some information about the state of the stack into permanent storage, so that a maintainer can look at it later and determine what went wrong.

These functions return information about stack frames. Each stack frame is identified by a *frame*, represented as a Lisp locative pointer. In order to use any of these functions, you need to have appropriate environment bindings set up. The macro **dbg:with-erring-frame** both sets up the environment properly and provides a frame pointer to the stack frame that got the error. Within the body of that macro, use the appropriate functions to move up and down stack frames; these functions take a frame pointer and return a new frame pointer by following links in the stack.

These frame-manipulating functions are actually *subprimitives*, even though they do not have a % sign in their name. Given an argument that is not a frame pointer, they stand a good chance of crashing the machine. Use them with care.

The functions that return new frame pointers work by going to the *next frame* or the *previous frame* of some category. "Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack). "Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; toward the base of the stack).

These functions assume three categories of frames:

- *Active* frames are procedures that are currently running (or active) on the stack.

- *Interesting active* frames include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as frames for **eval**, **apply**, **funcall**, **let**, and other basic Lisp special forms. The list of such functions is the value of the system constant **dbg:*uninteresting-functions***.

- *Open* frames include all the active frames, as well as frames that are still under construction, for functions whose arguments are still being computed. Open frames and active frames are synonymous on the 3600 series.

Functions for Examining Stack Frames

The functions in this section all take a frame pointer, *frame*, as an argument. For functions that indicate a direction on the stack, using **nil** as the argument indicates the frame at the relevant end of the stack. For example, when you are using a function that looks up the stack, **nil** as the argument indicates the top-most stack frame.

Remember to use the functions in this section only within the context of the **dbg:with-erring-frame** macro.

dbg:with-erring-frame *(frame-var condition)* &body *body*
Sets up an environment with appropriate bindings for using the rest of the functions that examine the stack.

dbg:get-frame-function-and-args *frame*
Returns a list containing the name of the function for *frame* and the values of the arguments.

dbg:frame-next-active-frame *frame*
Returns a frame pointer to the next active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-interesting-active-frame *frame*
Returns a frame pointer to the next interesting active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-open-frame *frame*
Returns a frame pointer to the next open frame following *frame*, or **nil** if *frame* is the last open frame on the stack.

dbg:frame-previous-active-frame *frame*
Returns a frame pointer to the previous active frame before *frame*, or **nil** if *frame* is the first active frame on the stack.

dbg:frame-previous-interesting-active-frame *frame*
Returns a frame pointer to the previous interesting active frame before *frame*, or **nil** if *frame* is the first active frame on the stack.

dbg:frame-previous-open-frame *frame*
Returns a frame pointer to the previous open frame before *frame*, or **nil** if *frame* is the first open frame on the stack.

dbg:frame-next-nth-active-frame *frame* &optional (*count* **1**) *skip-invisible*
Returns a frame pointer to the next nth active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-nth-interesting-active-frame *frame* &optional (*count* **1**) *skip-invisible*
Returns a frame pointer to the next nth interesting active frame following *frame*, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-next-nth-open-frame *frame* &optional (*count* **1**) *skip-invisible*
Returns a frame pointer to the next nth open frame following *frame*, or **nil** if *frame* is the last open frame on the stack.

dbg:frame-out-to-interesting-active-frame *frame*
Returns a frame pointer to the out to interesting active frame, (this include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as the frames for **eval**, **zl:apply**, **funcall**, **let**, and other basic Lisp special forms. The list of such functions is the value of the

system constant, **dbg:*uninteresting-functions***, or **nil** if *frame* is the last active frame on the stack.

dbg:frame-active-p *frame*
Determines if **t** if the *frame* is active; returns **t** if it is, or **nil** otherwise.

dbg:frame-real-function *frame*
Returns either the function object associated with *frame*, or the value of self when the frame was the result of sending a message to an instance.

dbg:frame-total-number-of-args *frame*
Returns a number of arguments that were passed in *frame*.

dbg:frame-number-of-spread-args *frame* &optional (*type* **:supplied**)
Returns a number of "spread" arguments that were passed in *frame*. *type* requests more specific definition of the number.

dbg:frame-arg-value *frame arg-name-or-number* &optional *callee-context no-error-p*
Returns a value of the *n*th argument to *frame*. Returns a second value, which is a locative pointer to the word in the stack that holds the argument. If *n* is out of range, the function takes action based on *no-error-p*; if the *no-error-p* is **nil**, it signals an error, otherwise it returns **nil**.

dbg:frame-number-of-locals *frame*
Returns a number of local variables allocated for *frame*.

dbg:frame-local-value *frame local-name-or-number* &optional *no-error-p*
Returns a value of the *n*th local variable in *frame*. Returns a second value, which is a locative pointer to the word in the stack that holds the local variable.

dbg:frame-self-value *frame* &optional *instance-frame-only*
Returns a value of **self** in *frame*, or **nil** if **self** does not have a value.

dbg:frame-real-value-disposition *frame*
Returns a symbol indicating how the calling function is going to handle the values to be returned by this frame. If the calling function just returns the values to its caller, then the symbol indicates how the final recipient of the values is going to handle them.

dbg:print-function-and-args *frame* &optional *show-pc-p show-source-file-p show-local-if-different*
Prints the name of the function executing in *frame* and the names and values of its arguments, in the same format as the Debugger uses. If *show-pc-p* is true, the program counter value of the frame, relative to the beginning of the function, is printed in octal.

dbg:print-frame-locals *frame local-start* &optional (*indent* **0**) *n-args-and-locals*
Prints the names and values of the local variables of *frame*.

Signalling Conditions

Signalling Mechanism

The following functions and macros invoke the signalling mechanism, which finds and invokes a handler for the condition.

error
signal
cerror
signal-proceed-case

Zetalisp Note: **zl:ferror** and **zl:fsignal** are the Zetalisp commands for invoking the signalling mechanism.

Finding a Handler

The signalling mechanism finds a handler by inspecting five lists of handlers in the following order:

1. It first looks down the list of *bound* handlers, which are set up by **condition-bind**, **condition-case**, and **condition-call** forms.

2. Next, it looks down the list of *default* handlers, which are set up by **condition-bind-default**.

3. Next, it looks down the list of *global* handlers, which are set up by **define-global-handler**.

4. Next, it looks down the list of *interactive* handlers. This list normally contains only one handler, which enters the Debugger if the condition is based on **dbg:debugger-condition**, and declines to handle it otherwise.

5. Finally, it looks down the list of *restart* handlers, which are set up by **error-restart**, **error-restart-loop**, and **catch-error-restart**. See the section "Default Handlers and Complex Modularity". See the section "Restart Handlers".

If it gets to the end of the last list without finding a willing handler, one of two things happen:

- **signal** returns **nil** when both of the following are true:
  - The condition is signalled with **signal**, **cerror**, **signal-proceed-case**, or **zl:fsignal**.

- The condition object is not an instance of a condition based on **error**.

- Otherwise, the Debugger assumes control.

The signalling mechanism checks each handler to see if it is willing to handle the condition. Some handlers have the ability to decline to handle the condition, in which case the signalling mechanism keeps searching. It calls the first willing handler it finds.

As we have seen, the signalling mechanism searches for handlers in a specific order. It looks at all the bound handlers before any of the default handlers, and all of the default handlers before any of the restart handlers. Thus, it tries any **condition-bind** handler before any handler bound by **condition-bind-default**, even though the **condition-bind-default** is within the dynamic scope of the **condition-bind**. Similarly, it considers a **condition-bind** handler before an **error-restart** handler, even when the **error-restart** handler was bound more recently. See the section "Default Handlers and Complex Modularity".

While a bound or default handler is executing, that handler and all handlers inside it are removed from the list of bound or default handlers. This is to prevent infinite recursion when a handler signals the same condition that it is handling, as in the following simplistic example:

```
(condition-bind ((error #'(lambda (x) (error "foo"))))
  (error "foo"))
```

If you want recursion, the handler should bind its own condition.

Signalling Simple Conditions

If a simple condition or a debugger condition not based on **error** is signalled, the signalling mechanism searches for a handler on the bound handler and default handler lists. When it finds one, it invokes it. Otherwise, the signalling mechanism checks for a global handler and invokes it if found. If there are no global handlers, the signalling mechanism checks for an interactive handler, invoking the first one it finds. If there are no interactive handlers, the first restart handler for that condition is invoked. If no restart handler for the condition is found, **signal** returns **nil**; **error** enters the Debugger.

Normally, there is only one interactive handler. This handler calls the Debugger if the condition is a debugger condition and not a simple condition.

Signalling Errors

In practice, if the **signal** function is applied to an error condition object, **signal** is very unlikely to return **nil**, because most processes contain a restart handler that handles all error conditions. The function at the base of the stack of most processes contains a **catch-error-restart** form that handles **error** and **sys:abort**. Thus, if you are in the Debugger as a result of an error, you can always use `ABORT`. The restart handler at the base of the stack always handles **sys:abort** and either terminates or restarts the process.

Restriction Due to Scope

A condition must be signalled only in the environment in which the event that it represents took place, to insure that handlers run in the proper dynamic environment. Therefore, you cannot signal a condition object that has already been signalled once. In particular, when you are writing a handler, you cannot have that handler signal its condition argument. Similarly, if a condition object is returned by some program (such as the **open** function given **nil** for the **:error** keyword), you cannot signal that object.

It is not correct to pass on the condition by signalling the handler's condition argument. This is incorrect:

```
(defun condition-handler (condition)
    (if something (throw ...) (signal condition)))
```

Instead you should do this:

```
(defun condition-handler (condition)
    (if something (throw ...) nil))
```

or this:

```
(defun condition-handler (condition)
    (if something (throw ...) (signal 'some-other-condition)))
```

Condition-Checking and Signalling Functions and Variables

signal *flavor* &rest *init-options*
: This is the primitive function for signalling a condition. Creates a new condition object of the specified *flavor* and signals it.

error {*format-string* &rest *format-args*} or {*condition* &rest *init-options*} or {*condition-object*}
: Signals a condition that is not proceedable. In the simplest case, signals a **zl:ferror** condition. If no handler for the condition exists, the debugger assumes control whether or not the object is an error object. In its most advanced form **error** is called with a single argument, *condition-object* and *init-options* is ignored. **error** never returns to its caller. If called with *condition* &rest *init-options,* creates a condition of type *condition* and signals it.

cerror *optional-condition-name continue-format-string error-format-string* &rest *args*
: Signals a proceedable error and enters the Debugger.

warn *optional-options optional-condition-name format-string* &rest *format-args*
: Prints a warning message and does not enter the debugger if ***break-on-warnings** is **nil**, otherwise enters the debugger, prints the message, and allows the user to proceed.

break-on-warnings

If value of this variable is **nil**, **warn** prints its warning message without entering the Debugger; if the value is not **nil**, **warn** enters the Debugger and prints the warning message. Default is **nil**.

catch-error *form* &optional (*printflag* **t**)

Evaluates *form*, trapping all errors. If the value of *printflag* is not **nil** and an error occurs during evaluation, the function prints an error message and returns **t**. If no error occurred, the value of *form* and **nil** are returned. *form* is not evaluated for multiple values.

make-condition *condition-name* &rest *init-options*

Creates a condition object of the specified *condition-name* with the specified *init-options*. This object can then be signalled by passing it to **signal** or **error**.

errorp *thing* Returns **t** if *thing* is an error object, and **nil** otherwise.

check-type *place type* &optional *(type-string '***nil***)*

Signals a proceedable error if the contents of *place* are not of the desired type. Accepts replacement value for *place*.

assert *test-form* &optional *references format-string* &rest *format-args*

Signals a proceedable error if the value of *test-form* is **nil**. Accepts replacement values for variables in *test-form*.

ccase *object* &body *body*

"Continuable exhaustive case." Like **case**, selects one of its clauses for execution by comparing a value to various constants, but does not allow an explicit **t** clause. Signals a proceedable error if no clause is satisfied; accepts replacement value for *object*.

ecase *object* &body *body*

"Exhaustive case," or "error-checking case." Like **case**, selects one of its clauses for execution by comparing a value to various constants, but does not allow an explicit **t** clause. Signals an error if no clause is satisfied. It is not permissible to continue from this error.

ctypecase *object* &body *body*

"Continuable exhaustive type case." Like **typecase**, selects one of its clauses by examining the type of an object, but does not allow an explicit **otherwise** or **t** clause. Signals a proceedable error if no clause is satisfied. Accepts replacement value for *object*.

etypecase *object* &body *body*

"Exhaustive type case," or "error-checking type case." Like **typecase**, but does not allow an explicit **otherwise** or **t** clause.

Signals an error if no clause is satisfied. It is not permissible to continue from this error.

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:fsignal *format-string* &rest *format-args*
Signals **dbg:proceedable-ferror**. This is a simple function signalling when you do not care to use a particular condition. Use the Common Lisp function **cerror**.

zl:ferror *format-string* &rest *format-args*
This is a simple function for signalling when you do not care what the condition is. Use the Common Lisp function, **error**.

zl:parse-ferror *format-string* &rest *format-args*
Signals an error of flavor **zl:parse-ferror**.

zl:check-arg *arg-name predicate-or-form type-string*
Checks arguments to make sure they are valid. Accepts replacement value for *arg-name*.

zl:check-arg-type *arg-name type* &optional *type-string*
A useful variant of **zl:check-arg**.

zl:argument-typecase *arg-name* &body *clauses*
This is a hybrid of **zl:typecase** and **zl:check-arg-type**. Automatically generates an **otherwise** clause that signals an error. Accepts replacement value for *arg-name*.

Default Handlers and Complex Modularity

When more than one handler exists for a condition, which one should be invoked? The signalling mechanism has an elaborate rule, but in practice, it usually invokes the innermost handler. See the section "Finding a Handler". "Innermost" is defined dynamically and thus means "the most recently bound handler".

This decision is made on the basic principle of modularity and referential transparency: A function should behave the same way, regardless of what calls it. Therefore, whether a handler bound by a function gets invoked should not depend on what is going on with that function's callers.

For example, suppose function **a** sets up a handler to deal with the **fs:file-not-found** condition, and then calls procedure **b** to perform some service for it. Now, unbeknownst to **a**, **b** sometimes opens a file, and **b** has a condition handler for **fs:file-not-found**. If **b**'s file is not found, **b**'s handler handles the error rather than **a**'s. This is as it should be, because it should not be visible to **a** that **b** uses a file

(this is a hidden implementation detail of **b**). **a**'s unrelated condition handler should not meddle with **b**'s internal functioning. Therefore, the signalling mechanism follows a basic inside-to-outside searching rule.

Sometimes a function needs to signal a condition but still handle the condition itself if none of its callers handles it. On first encounter, this need seems to require an outside-to-inside searching rule instead of the inside-to-outside searching rule mandated by modularity considerations. How can you circumvent the rules to allow a function to handle something only if no outer function handles it?

Several strategies are available for dealing with this. Genera provides several mechanisms in order to allow experimentation and flexibility.

- The simplest solution is to provide a proceed type for proceeding from the Debugger. That is, your program signals an error to allow callers to handle the condition. If none of them handles it, the Debugger assumes control. Provided the user decides to use the proceed type, your program then gets to handle the condition. If what your program wanted to do was to prompt the user anyway, this might be the right thing. This is most likely true if you think that a program error is probably happening and the user might want to be able to trace and manipulate the stack using the Debugger.

- Another simple solution is to signal a condition that is not an error. **signal** returns **nil** when no handler is found, and your program can take appropriate action.

- Use **condition-bind-default** to create a handler on the default handler list. The signalling mechanism searches this list only after searching through all regular bound handlers. One drawback of this scheme is that it works only to one level. If you have three nested functions, you cannot get outside-to-inside modularity for all three, because only two lists exist, the bound list and the default list. This facility is probably good enough for some applications however.

- Use **dbg:condition-handled-p** to determine whether a handler has been bound for a particular flavor. This has the advantage that it works for any number of levels of nested handler, instead of only two. One disadvantage is that it can return **:maybe**, which is ambiguous.

The simple solutions work only if your program is doing the signalling. If some other program is signalling a condition, you cannot control whether the condition is an error condition or whether it has any proceed types; you can only write handlers.

Restart Handlers

One way to handle an error is to restart at some earlier point in the program that got the error. A program can specify points where it is safe or convenient for it to be restarted should a condition signal occur during processing of a function. The basic special form for doing this is **error-restart**. The following example is taken

from the system code:

```
(defun connect (address contact-name
                &optional (window-size default-window-size)
                (timeout (* 10. 60.))
                &aux conn real-address (try 0))
  (error-restart (sys:connection-error
                   "Retry connection to ~A at ~S with longer timeout"
                   address contact-name)
    forms...))
```

This code fragment evaluates *forms* and returns the final value(s) if successful. If the Debugger assumes control as a result of a **sys:connection-error** condition, the user is given the opportunity of restarting the program. The Debugger's prompt message would be something like this:

```
s-A:  Retry connection to SCRC at FILE 1 with longer timeout
```

If the user were to press `s-A` at this point, the forms implementing the connection would be evaluated again. That is, the body of the **error-restart** would be started again from the beginning.

Two variations on this basic paradigm are provided. **error-restart-loop** is an infinite loop version of **error-restart**. It always starts over regardless of whether a condition has been signalled. **catch-error-restart** never restarts, even when a condition is signalled. Instead it always returns, returning either the values from the body (if successful) or **nil** if a condition signal occurred.

catch-error-restart is the most primitive version of this control structure. The other two are built from it. It too has a conditional variant, **catch-error-restart-if**, for binding a restart handler conditionally.

A common paradigm is to use one of these forms in the command loop of an interactive program, with *condition-flavor* being the list **(error sys:abort)**. This way, if an unhandled error occurs, the user is offered the option of returning to the command loop, and the `ABORT` key returns to the command loop. Which form you use depends on the nature of your command loop.

Restart Functions

The use of "error-" in the names of these functions has no real significance. They could have been called **cl:condition-restart cl:condition-restart-loop** and so on, because they apply to all conditions.

error-restart *(flavors description* &rest *args)* &body *body*

The basic special form for a program to specify safe restart points. Establishes a restart handler for *flavors* and then evaluates *body*. If restart handler is not invoked, **error-restart** returns the values produced by the last form in *body*. If restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart** form, and execution of *body* starts all over again.

error-restart-loop *(flavors description* &rest *args)* &body *body*
Establishes a restart handler for *flavors* and then evaluates *body*. If the handler is not invoked, **error-restart-loop** evaluates *body* again and again, in an infinite loop. Use the **return** function to leave the loop. This mechanism is useful for interactive top levels. If a condition is signalled during the execution of *body* and the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart-loop** form and execution of *body* is started all over again.

catch-error-restart *(flavors description* &rest *args)* &body *body*
Establishes a restart handler for *flavors* and then evaluates *body*. If the handler is not invoked, **catch-error-restart** returns the values produced by the last form in *body*, and the restart handler disappears. If a condition is signalled during the execution of body and the restart handler is invoked, control is thrown back to the dynamic environment of the **catch-error-restart** form. In this case, **catch-error-restart** also returns **nil** as its first value and something other than **nil** as its second value.

catch-error-restart-if *cond (flavors description* &rest *args)* &body *body*
The conditional variant of **catch-error-restart**. This form establishes its restart handler conditionally.

Invoking Restart Handlers Manually

Function **dbg:invoke-restart-handlers** searches the list of restart handlers to find a restart handler. The first handler it finds to handle the condition is invoked. The function returns **nil** if no appropriate restart handler is found.

dbg:invoke-restart-handlers can be called by handlers set up by **condition-bind** or **condition-bind-default**. The *object* argument should be the condition object passed to the handler. The handler calls this function to bypass the interactive handlers list, letting the innermost restart handler handle the condition. A program that wants to attempt to continue with a computation in the presence of errors might find this useful. For example, it could be used to support batch-mode compilation, with the user away from the console.

Proceeding

In some situations, execution can proceed past the point at which a condition was signalled. Events for which this is the case are called *proceedable conditions*. Some external agent makes the decision about whether it is reasonable to proceed after repairing the original problem. The agent is either a **condition-bind** handler or the user operating the Debugger.

In general, many different ways are available to proceed from a particular condition. Each way is identified by a *proceed type*, which is represented as a symbol. Condition objects created with **error** instead of **signal** do not have any proceed types.

Protocol for Proceeding

In order for proceeding to work, two conceptual agents must agree:

- The programmer who wrote the program that signals the condition.
- The programmer who wrote the **condition-bind** handler that decided to proceed from the condition, or else the user who told the Debugger to proceed.

The signaller signals the condition and provides the various proceed types. The handler chooses from among the proceed types to make execution proceed.

Each agent has certain responsibilities to the other; each must follow the protocol described below to make sure that any handler interacts correctly with any signaller. The following description should be considered a two-part protocol that each agent must follow in order to communicate correctly with the other.

In very simple cases, the signaller can use **cerror**, which does not require any new flavor definitions.

In all other cases, the signaller signals the condition using **signal** or **signal-proceed-case**. The signaller also defines a condition flavor with at least one method to handle a proceed type. New proceed types are always defined by adding a new case to the **sys:proceed** method (which is defined to use **:case** method combination) to the condition flavor. The method must always return values rather than throwing.

In **:case** method combination, the second argument to the **sys:proceed** method is like a subsidiary message name, causing a further dispatch, just as the original message name caused a primary dispatch. See the section "**:case** Method Combination Type".

Here's an example from the system:

```
(defmethod (sys:proceed sys:subscript-out-of-bounds  :new-subscript)
           (&optional (sub (prompt-and-read :number
                                            "Subscript to use instead: ")))
  "Supply a different subscript."
  (values :new-subscript sub))
```

This code fragment creates a proceed type called **:new-subscript** for the condition flavor **sys:subscript-out-of-bounds**.

To proceed from a condition, a handler function calls the **sys:proceed** generic function with one or more arguments. The first argument is the *condition* object. The second argument is the proceed type, and any remaining arguments are the arguments for that proceed type.

The condition flavor defined by the program signalling the error defines the proceed types that are available to **sys:proceed** for a particular condition. You can also define a method that creates a new proceed type with the **:case** method combination.

All of the arguments to a **sys:proceed** method must be optional arguments. The **sys:proceed** method should provide default values for all its arguments. One useful way of doing this is to prompt a user for the arguments using the ***query-io*** stream. The example uses **prompt-and-read**. If all the optional arguments are supplied, the **sys:proceed** method must not do any input or output using ***query-io***.

This facility has been defined assuming that **condition-bind** handlers would supply all the arguments for the method themselves. The Debugger runs this method and does not supply arguments, relying on the method to prompt the user for the arguments.

As in the example, the method should have a documentation string as the first form in its body. The **dbg:document-proceed-type** generic function for a proceedable condition object displays the string. This string is used by the Debugger as a prompt to describe the proceed type. For example, the subscript example might result in the following Debugger prompt:

```
s-A: Supply a different subscript
```

The string should be phrased as a one-line description of the effects of proceeding from the condition. It should not have any leading or trailing newline characters. (You can use as models the messages that the Debugger prints out to describe the effects of the `s-` commands, if you are interested in stylistic consistency.)

Sometimes a simple fixed string is not adequate. You can provide a piece of Lisp code to compute the documentation text at run time by providing your own method for **dbg:document-proceed-type**. This method definition takes the following form:

```
(defmethod (dbg:document-proceed-type condition-flavor proceed-type)
           (stream)
  body...)
```

The body of the method should print documentation for *proceed-type* of *condition-flavor* onto *stream*.

The body of the **sys:proceed** method can do anything it wants. In general, it tries to repair the state of things so that execution can proceed past the point at which the condition was signalled. It can have side-effects on the state of the environment; it can return values so that the function that called **signal** can try to fix things up; or it can do both. Its operation is invisible to the handler; the signaller is free to divide the work between the function that calls **signal** and the **sys:proceed** method, as it sees fit. When the **sys:proceed** method returns, **signal** returns all of those values to its caller. That caller can examine them and take action accordingly.

The meaning of these returned values is strictly a matter of convention between the **sys:proceed** method and the function calling **signal**. It is completely internal to the signaller and invisible to the handler. By convention, the first value is usually the name of a proceed type. See the section "Signallers".

A **sys:proceed** method can return a first value of **nil** if it declines to proceed from the condition. If a **nil** returned by a **sys:proceed** method becomes the return value for a **condition-bind** handler, this signifies that the handler has declined to handle the condition, and the condition continues to be signalled. When the **sys:proceed** function is called by the Debugger, the Debugger prints a message saying that the condition was not proceeded, and it returns to its command level. This might be used by an interactive **sys:proceed** method that gives the user the opportunity either to proceed or to abort; if the user aborts, the method returns **nil**. Returning **nil** from a **sys:proceed** method should not be used as a substitute for detecting earlier (such as when the condition object is created) that the proceed type is inappropriate for that condition.

Proceed Type Functions

By default, condition objects have to handle all proceed types defined for the condition flavor. Condition objects can be created that handle only some of the proceed types for their condition flavor. When the signaller creates the condition object (with **signal** or **make-condition**), it can use the **:proceed-types** init option to specify which proceed types the object accepts. The value of the init option is a list of keyword symbols naming the proceed types.

```
(signal 'my-condition :proceed-types '(:abc))
```

The **dbg:proceed-types** generic function for a condition object returns a list of keywords for the proceed types that the object is prepared to handle.

The **dbg:proceed-type-p** generic function examines the list of valid proceed types to see whether it contains a particular proceed type.

A condition flavor might also have an **:init** daemon that could modify its **dbg:proceed-types** instance variable.

Proceeding with condition-bind Handlers

Suppose the handler is a **condition-bind** handler function. Just to review, when the condition is signalled, the handler function is called with one argument, the condition object. The handler function can throw to some tag, return **nil** to say that it doesn't want to handle the condition, or try to proceed the condition.

The handler must not attempt to proceed using an invalid proceed type. It must determine which proceed types are valid for any particular condition object. It must do this at run time because condition objects can be created that do not handle all of the proceed types for their condition flavor. See the init option **(flavor:method :proceed-types condition)**.

In addition, condition objects created with **error** instead of **signal** do not have any proceed types. The handler can use the **dbg:proceed-types** and **dbg:proceed-type-p** functions to determine which proceed types are available.

To proceed from a condition, a handler function calls **sys:proceed** on the condition object with one or more additional arguments. The first additional argument is the

proceed type (a keyword symbol) and the rest are the arguments for that proceed type. All of the standard proceed types are documented with their condition flavors. Thus, the programmer writing the handler function can determine the meanings of the arguments. The handler function must always pass all of the arguments, even though they are optional.

Calling **sys:proceed** should be the last thing the handler does. It should then return immediately, propagating the values from the **sys:proceed** method back to its caller. Determining the meaning of the returned values is the business of the signaller only; the handler should not look at or do anything with these values.

Proceed Type Names

Any symbol can be used as the name of a proceed type, although using keyword symbols is conventional. The symbols **:which-operations** and **:case-documentation** are not valid names for proceed types because they are treated specially by the **:case** flavor combination. Do not use either of these symbols as the name of a proceed type when you create a new condition flavor.

Signallers

Signallers can use the **signal-proceed-case** special form to signal a proceedable condition. **signal-proceed-case** assumes that the first value returned by every proceed type is the keyword symbol for that proceed type. This convention is not currently enforced.

Example of User-Defined Error Flavor

Here is an example of an application program defining its own error flavor, which indicates that an editor buffer was not found, with two proceed types: One proceed type asks the user to supply a new buffer name; the other offers to create a new buffer.

```
(defflavor buffer-not-found ((pathname nil) (name nil)
                             defaults create-p load-p) (error)
  :initable-instance-variables
  :readable-instance-variables)

(defmethod (dbg:report buffer-not-found) (stream)
  (format stream "The buffer ~:[named~;for pathname~]
                              /"~A/" was not found."
                              (null name) (or name pathname)))
```

```
(defmethod (sys:proceed buffer-not-found :choose-another-buffer)
           (&optional new-name)
  (if (not new-name)
      (setq new-name
            (cond (name (prompt-and-read ':string
               "Supply another buffer name to use instead of ~A:  "
                                         name))
                  (t (prompt-and-read ':pathname
                                      "Supply another pathname to
                    use instead of ~A~ ~:[~; (Default = ~A)~]:  "
                                      name defaults
                                      (fs:default-pathname
                                        defaults)))))
      (find-or-create-buffer nil (and name new-name)
                             (and pathname (fs:parse-pathname
                                             new-name))
                             nil nil
                             defaults
                             create-p load-p nil)))

(defmethod (dbg:document-proceed-type buffer-not-found
                                      :choose-another-buffer)
           (stream)
  (format stream
          "Supply another ~:[buffer name~;pathname~]
             to use instead."
          (null name)))

(defmethod (sys:proceed buffer-not-found :create-buffer) ()
  (find-or-create-buffer nil name pathname nil nil
                         defaults t load-p nil))

(defmethod (dbg:document-proceed-type buffer-not-found
                                      :create-buffer)
           (stream)
  (if (null name)
      (format stream "Create a buffer for file /"~A/"
                          ~:[~; and load it from the file~]."
              pathname load-p)
      (format stream "Create a buffer named /"~A/"." name)))

(compile-flavor-methods buffer-not-found)
```

Issues for Interactive Use

Tracing Conditions

The variable **sys:trace-conditions** is provided for debugging purposes only. It lets you trace the signalling of any condition so that you can figure out what conditions are being signalled, and by what function. You can set this variable to **error** to trace all error conditions, for example, or you can be more specific.

You can also customize the error message displayed by the Debugger by binding the variable **sys:error-message-hook** to a function that prints what you want. When the Debugger finds the value of **sys:error-message-hook**, to be non-**nil**, it **funcall**s (applies the function) with no arguments, and displays the results at the end of its error message display.

Breakpoints

The functions **breakon** and **unbreakon** can be used to set breakpoints in a program. They use the encapsulation mechanism like, **trace** and **advise**, to force the function to signal a condition when it is called.

Breakpoint Functions

breakon &optional *function* (*condition* **t**)

Sets a breakpoint for the *function*. *condition* can be used for making a conditional breakpoint. It is evaluated when the function is called. If it returns **nil**, the function call proceeds without signalling anything. *condition* is evaluated in the dynamic environment of the function call. You can inspect the arguments of *function* by looking at the variable **arglist**.

unbreakon &optional *function* (*condition* **t**)

Turns off a breakpoint set by **breakon**. If *function* is not provided, all breakpoints set by **breakon** are turned off. If *condition* is provided, it turns off only that condition, leaving any others. If *condition* is not provided, the entire breakpoint is turned off for that function. See the section "Encapsulations".

Calling a function for which a breakpoint is set signals a condition with the following message:

```
Break on entry to function name
```

It provides a **:no-action** proceed type, which allows the function entry to proceed. The "trap on exit" bit is set in the stack frame of the function call, so that when the function returns, or is thrown through, another condition is signalled. Similarly, the "Break on exit from marked frame" message and the **:no-action** proceed type are provided, allowing the function return to proceed.

Debugger Bug Reports

The :Mail Bug Report (c-M) command in the Debugger sends a bug report, creating a new process, which, by default, sends the bug report to the BUG-GENERA mailing list. Also by default, the mail-sending text buffer initially contains a standard set of information dumped by the Debugger. You can control this behavior for your own condition flavors. You can control the mailing list to which the bug report is sent by defining your own primary method for the following message. You can also control the character style of the system You.

To control the initial contents of the mail-sending buffer, alter the handling of the following message, either by providing your own primary method to replace the default message, or by defining a **:before** or **:after** daemon to add your own specialized information before or after the default text.

Debugger Bug Report Functions

dbg:bug-report-recipient-system *condition*
: Returns the mailing list to which to send the bug report mail.

dbg:bug-report-description *condition stream nframes*
: Prints out the initial contents of the mail-sending buffer.

dbg:*character-style-for-bug-mail-prologue*
: Controls the character style for initial contents of the mail-sending buffer. The default is **'(nil nil :tiny)**, which makes the information compact.

Debugger Special Commands

When the Debugger assumes control because an error condition was signalled and not handled, it offers the user various ways to proceed or to restart. Sometimes you want to offer the user other kinds of options. In the system, the most common example of this occurs when you forget to type a package prefix. It signals a **sys:unbound-symbol** error and offers to let you use the symbol from the right package instead. This is neither a proceed type nor a restart-handler; it is a Debugger special command.

You can add one or more special commands to any condition flavor. For any particular instance, you can control whether to offer the special command. For example, the package-guessing service is not offered unless some other symbol with the same print name exists in a different package. Special commands are called only by the Debugger; **condition-bind** handler functions never see them.

Special commands provide the same kind of functionality that a **condition-bind** handler does. There is no reason, for example, that the package-prefix service could not have been provided by **condition-bind**. It is only a matter of convenience to have it in a special command.

To add special commands to your condition flavor, you must mix in the flavor **dbg:special-commands-mixin**, which provides both the instance variable

dbg:special-commands and several method combinations. Each special command to a particular flavor is identified by a keyword symbol, just the same way that proceed types are identified. You can then create handlers for any of the following messages:

Debugger Special Command Functions

dbg:special-command *condition* &rest *per-command-args*
Sent when the user invokes the special command.

dbg:document-special-command *condition special-command*
Prints the documentation of *special-command*.

dbg:initialize-special-commands *condition*
The Debugger calls this after it prints the error message. The methods for this generic function can remove items from the list **dbg:special-commands** in order to decide not to offer these special commands.

Special Keys

The system normally handles the ABORT and SUSPEND keys so that ABORT aborts what you are doing and SUSPEND enters a breakpoint. Without a CONTROL modifier, such a keystroke command takes effect only when the process reads the character from the keyboard; with the CONTROL modifier, a keystroke command takes effect immediately, regardless of what the process is doing. The META modifier means "do it more strongly"; m-ABORT resets the process entirely, and m-SUSPEND enters the Debugger instead of entering a simple read-eval-print loop.

A complete and accurate description of what these keys do requires a discussion of conditions and the Debugger.

With no CONTROL modifier, ABORT and SUSPEND are detected when your process tries to do input from the keyboard (typically by doing an input stream operation such as **:tyi** on a window). Therefore, if your process is computing or waiting for something else, the effects of the keystrokes are deferred until your process tries to do input.

With a CONTROL modifier, ABORT and SUSPEND are intercepted immediately by the Keyboard Process, which sends your process an **:interrupt** message. Thus, it performs the specified function immediately, even if it is computing or waiting.

ABORT
Prints the following string on the ***terminal-io*** stream, unless it suspects that output on that stream might not work.

```
[Abort]
```

It then signals a **(process-abort *current-process*)**, which is a simple condition. Programs can set up bound handlers for **sys:abort**, although most do not. Many programs set up restart

handlers for **sys:abort**; most interactive programs have such a handler in their command loops. Therefore, ABORT usually restarts your program at the innermost command loop. Inside the Debugger, ABORT has a special meaning.

m-ABORT — Prints the following string on the ***terminal-io*** stream, unless it suspects that output on that stream might not work.

```
[Abort all]
```

It then sends your process a **:reset** message, with the argument **:always**. This has nothing to do with condition signalling. It just resets the process completely, unwinding its entire stack. What the process does after that depends on what kind of process it is and how it was created: it might start over from its initial function, or it might disappear. See the section "How the Scheduler Works".

SUSPEND — Calls the **zl:break** function with the argument **zl:break**. This has nothing to do with condition signalling. Pressing the RESUME key causes the process to resume execution. See the special form **zl:break**.

m-SUSPEND — Causes the Debugger to assume control without signalling any condition. The RESUME key in the Debugger causes the Debugger to return and the process to resume what it was doing.

Several techniques are available for overriding the standard operation of ABORT and SUSPEND when they are being used with modifier keys.

- For using these keys with the CONTROL modifier, use the asynchronous character facility. See the section "Asynchronous Characters".

- Defining your own hook function and binding **tv:kbd-tyi-hook** to it also overrides the interception of these characters with no CONTROL modifier. See the section "Windows as Input Streams".

At the Debugger command loop, ABORT is the same as the Debugger :Abort (c-Z) command. It throws directly to the innermost restart handler that is appropriate for either the current error or the **sys:abort** condition.

When the Debugger assumes control, it displays a list of commands appropriate to the current condition, along with key assignments for each. Proceed types come first, followed by special commands, followed by restart handlers. One alphabetic key with the SUPER modifier is assigned to each command on the list. In addition, ABORT is always assigned to the innermost restart handler that handles **sys:abort** or the condition that was signalled; RESUME is always assigned to the first proceed type in the **dbg:proceed-types** list. See the section "Proceed Type Functions".

If RESUME is not otherwise used, it invokes the first error restart that does not handle **sys:abort**. When you enter the Debugger with m-SUSPEND, RESUME resumes the process.

You can customize the Debugger, assigning certain keystrokes to certain proceed types or special commands, by setting the variables listed below in your init file:

Debugger Special Key Variables

dbg:*proceed-type-special-keys*

The value of this variable should be an association list associating proceed types with characters. When an error supplies any of these proceed types, the Debugger assigns that proceed type to the specified key.

dbg:*special-command-special-keys*

The value of this variable should be an association list associating proceed types with characters. When an error supplies any of these special commands, the Debugger assigns that special command to the specified key.

Condition Flavors Reference

A condition object is an instance of any flavor built out of the **condition** flavor. An error object is an instance of any flavor built out of the **error** flavor. The **error** flavor is built out of the **dbg:debugger-condition** flavor, which is built out of the **condition** flavor. Thus, all error objects are also condition objects.

Every flavor of condition that is instantiated must handle the **dbg:report** generic function. (Flavors that just define sets of conditions need not handle it). This message takes a stream as its argument and prints out a textual message describing the condition on that stream. The printed representation of a condition object is like the default printed representation of any instance when slashifying is turned on. However, when slashifying is turned off (by **princ** or the **~A format** directive), the printed representation of a condition object is its **dbg:report** method. Example:

```
(condition-case (co)
     (open "f:>a>b.c")
  (fs:file-not-found
     (prin1 co)))    prints out  #<QFILE-NOT-FOUND 33712233>

(condition-case (co)
     (open "f:>a>b.c")
  (fs:file-not-found
     (princ co)))    prints out  The file was not found
For F:>a>b.c
```

Generic Functions and Init Options

These functions can be applied to any condition object. They are handled by the basic **condition** flavor, on which all condition objects are built. Some particular

condition flavors handle other methods; those are documented along with the particular condition flavors in another section. See the section "Standard Conditions".

Basic Condition Methods and Init Options

dbg:document-proceed-type *condition proceed-type stream*
Prints out a description of what it means to proceed, using the given *proceed-type*, from this condition, on *stream*.

dbg:proceed-type-p *condition proceed-type*
Returns **t** if *proceed-type* is one of the valid proceed types of this condition object. Otherwise, returns **nil**.

dbg:proceed-types *condition*
Returns a list of all the valid proceed types for this condition.

dbg:set-proceed-types *condition new proceed-types*
Sets the list of valid proceed types for this condition to *new-proceed-types*.

dbg:special-commands *condition*
Returns a list of all Debugger special commands for this condition.

:proceed-types This init option defines the set of proceed types to be handled by this instance. *proceed-types* is a list of proceed types (symbols); it must be a subset of the set of proceed types understood by this flavor. If this option is omitted, the instance is able to handle all of the proceed types understood by this flavor in general, but by passing this option explicitly, a subset of acceptable proceed types can be established. This is used by **signal-proceed-case**.

dbg:special-command-p *condition special-command*
Returns **t** if *special-command* is a valid debugger special command for the condition object, *condition*. Returns **nil** otherwise.

dbg:report *condition stream*
Prints the text message associated with this object onto *stream*.

dbg:report-string *condition*
Returns a string containing the report message associated with this object. (sends **dbg:report** to the object.)

Standard Conditions

This section presents the standard condition flavors provided by the system. Some of these flavors are the flavors of actual condition objects that get instantiated in response to certain conditions. Others never actually get instantiated, but are used to build other flavors.

In some cases, the flavor that the system uses to signal an error is not exactly the one listed here, but rather a flavor built on the one listed here. This often comes up when the same error can be reported by different programs that implement a generic protocol. For example, the condition signalled by a remote file-system stream when a file is not found is different from the one signalled by a local file-system stream; however, only the generic **fs:file-not-found** condition should ever be handled by programs, so that a program works regardless of what kind of file-system stream it is using. The exact flavors signalled by each file system are considered to be internal system names, subject to change without notice and not documented here.

Do not look at system source code to figure out the names of error flavors without being careful to choose the right level of flavor! Furthermore, take care to choose a flavor that can be instantiated if you try to signal a system-defined condition. For example, you cannot signal a condition object of type **fs:file-not-found**, because this is really a set of errors and this flavor does not handle the **dbg:report** message. If you were to implement your own file system and wanted to signal an error when a file cannot be found, it should probably have its own flavor built on **fs:file-not-found** and other flavors.

Choosing the appropriate condition to handle is a difficult problem. In general you should not choose a condition on the basis of the apparent semantics of its name. Rather, you should choose it according to the appropriate level of the condition flavor hierarchy. This holds particularly for file-related errors. See the section "File-System Errors".

There are six classes of standard conditions:

- Fundamental Conditions
- Lisp Errors
- File-system Errors
- Pathname Errors
- Network Errors
- Tape Errors

Individual classes, their base flavors, and the conditions built on them, are discussed below.

Fundamental Conditions

These conditions are basic to the functionality of the condition mechanism, rather than having anything to do with particular system errors.

Here is a summary list of fundamental conditions. More detailed discussion of each follows the listing.

- **condition**
- **dbg:debugger-condition**
- **error**
- **zl:ferror**

- **dbg:proceedable-ferror**
- **sys:no-action-mixin**
- **sys:abort**
- **zl:break**

condition *Flavor*

This is the basic flavor on which all condition objects are built. User-defined conditions are not normally built directly upon **condition**.

dbg:debugger-condition *Flavor*

This flavor is built on **condition**. It is used for entering the Debugger without necessarily classifying the event as an error. This is intended primarily for system use; users should normally build on **error** instead.

error *Flavor*

This flavor is built on **dbg:debugger-condition**. All flavors that represent errors, as opposed to debugger conditions or simple conditions, are built on this flavor.

zl:ferror *Flavor*

This is a simple error flavor for the **zl:ferror** function. Use it when you do not want to invent a new error flavor for a certain condition. Its only state information is an error message, normally created by the call to the **zl:ferror** function. It has two readable and initable instance variables **format-string** and **format-args**. The **zl:format** function is applied to these values to produce the **dbg:report** message.

dbg:proceedable-ferror *Flavor*

This is a simple error flavor for the **zl:fsignal** function. Use it when you do not want to invent a new error flavor for a certain condition, but you want the condition to be proceedable. Its only state information is an error message, created by the call to the **zl:fsignal** function. Its only proceed type is **:no-action**. Proceeding in this way does nothing and causes **zl:fsignal** (or **signal**) to return the symbol **:no-action**.

sys:no-action-mixin *Flavor*

This flavor can be mixed into any condition flavor to define a proceed type called **:no-action**. Proceeding in this way causes the computation to proceed as if no error check had occurred. The signaller might try the action again or might simply go on doing what it would have done. For example, **proceedable-ferror** is just **zl:ferror** with this mixin.

sys:abort *Flavor*

The ABORT key on the keyboard was pressed. This is a simple condition. When **sys:abort** is signalled, control is thrown straight to a restart handler without entering the Debugger.

Note: It is preferable to use (**process-abort *current-process***) instead of (**signal sys:abort**). See the section "Special Keys".

zl:break *Flavor*

This is the flavor of the condition object passed to the Debugger as a result of the m-BREAK command. It is never actually signalled; rather, it is a convention to ensure that the Debugger always has a condition when it assumes control. This is based on **dbg:debugger-condition**. See the section "Special Keys".

Lisp Errors

This section describes the conditions signalled for basic Lisp errors. All of the conditions in this section are based on the **error** flavor unless otherwise indicated.

Lisp errors include the following ten major groups:

- Base Flavor: **sys:cell-contents-error**
- Location Errors
- Base Flavor: **sys:arithmetic-error**
- Base Flavor: **sys:floating-point-exception**
- Miscellaneous System Errors Not Categorized by Base Flavor
- Function-Calling Errors
- Array Errors
- Eval Errors
- Interning Errors Based on **sys:package-error**
- Errors Involving Lisp Printed Representations

Base flavor: sys:cell-contents-error

This group includes the following errors:

- **sys:cell-contents-error**
- **sys:unbound-variable**
- **sys:unbound-symbol**
- **sys:unbound-closure-variable**
- **sys:unbound-instance-variable**
- **sys:undefined-function**
- **sys:bad-data-type-in-memory**

sys:cell-contents-error *Flavor*

All the kinds of errors resulting from finding invalid contents in a cell of virtual memory are built on this flavor. This represents a set of errors including the various kinds of unbound-variable errors, the undefined-function error, and the bad data-type error.

Proceed type	*Action*
:new-value	Takes one argument, a new value to be used instead of the contents of the cell.
:store-new-value	Takes one argument, a new value to replace the contents of the cell.
:no-action	If you have intervened and stored something into the cell, the contents of the cell can be reread.

sys:unbound-variable *Flavor*

All the kinds of errors resulting from unbound variables are built on this flavor. Because these are a subset of the "cell contents" errors, this flavor is built on **sys:cell-contents-error**. The **:variable-name** message returns the name of the variable that was unbound (a symbol).

sys:unbound-symbol *Flavor*

An unbound symbol (special variable) was evaluated. Some instances of this flavor provide the **:package-dwim** special command, which takes no arguments and asks whether you want to examine the value of various other symbols with the same print name in other packages. This proceed type is provided only if any such symbols exist in any other packages. (See also **dbg:*defer-package-dwim***.) This flavor is built on **sys:unbound-variable**. The proceed types from **sys:cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:unbound-closure-variable *Flavor*

An unbound closure variable was evaluated. This flavor is built on **sys:unbound-variable**. The proceed types from **sys:cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:unbound-instance-variable *Flavor*

An unbound instance variable was evaluated. The **:instance** message returns the instance in which the unbound variable was found. The proceed types from **sys:cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable**.

sys:undefined-function *Flavor*

An undefined function was invoked; that is, an unbound function cell was referenced. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types. The **:function-name** message returns the name of the function that was undefined (a function spec). This also provides **:package-dwim** service, like **sys:unbound-symbol**.

sys:bad-data-type-in-memory *Flavor*

A word with an invalid type code was read from memory. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types.

Message	*Value returned*
:address	virtual address, as a locative pointer, from which the word was read
:data-type	numeric value of the data-type tag field of the word
:pointer	numeric value of the pointer field of the word

Location Errors

This group includes the following errors:

- **sys:unknown-setf-reference**
- **sys:unknown-locf-reference**

sys:unknown-setf-reference *Flavor*

zl:setf did not find a **zl:setf** property on the **car** of the form. The **:form** message returns the form that **zl:setf** tried to operate on. This error is signalled when the **zl:setf** macro is expanded.

sys:unknown-locf-reference *Flavor*

locf did not find a **locf** property on the car of the form. The **:form** message returns the form that **locf** tried to operate on. This error is signalled when the **locf** macro is expanded.

Base Flavor: sys:arithmetic-error

This group includes the following errors:

- **sys:arithmetic-error**
- **sys:divide-by-zero**
- **sys:non-positive-log**
- **math:singular-matrix**

sys:arithmetic-error *Flavor*

Represents the set of all arithmetic errors. No condition objects of this flavor are actually created; any arithmetic error signals a more specific condition, built on this one. This flavor is provided to make it easy to handle any arithmetic error.

All arithmetic errors handle the **:operands** message. This returns a list of the operands in the operation that caused the error.

sys:divide-by-zero *Flavor*

Division by zero was attempted. This flavor is built on **sys:arithmetic-error**. The **:function** message returns the function that did the division.

sys:non-positive-log *Flavor*

Computation of the logarithm of a nonpositive number was attempted. This flavor is built on **sys:arithmetic-error**. The **:number** message returns the nonpositive number.

math:singular-matrix *Flavor*

A singular matrix was given to a matrix operation such as inversion, taking of the determinant, or computation of the LU decomposition. This flavor is built on **sys:arithmetic-error**.

Base flavor: sys:floating-point-exception

sys:floating-point-exception and the condition flavors based on it are designed to support IEEE floating-point standards. See the section "Numbers". By default, all IEEE traps are enabled, except for the inexact-result trap. See the special form **without-floating-underflow-traps**.

The trap handlers that signal these conditions from the system all cause pressing the RESUME key to mean "return the result that would have been returned if the trap had been disabled". For example, pressing RESUME on an overflow returns the appropriately signed infinity as the result. On an underflow it returns the denormalized (possibly zero) result.

This group includes the following errors:

- **sys:floating-point-exception**
- **sys:float-divide-by-zero**
- **sys:floating-exponent-overflow**
- **sys:floating-exponent-underflow**
- **sys:float-inexact-result**
- **sys:float-invalid-operation**
- **sys:float-invalid-compare-operation**

- **sys:negative-sqrt**
- **sys:float-divide-zero-by-zero**

sys:floating-point-exception *Flavor*

This is the base flavor for floating-point exceptional conditions. No condition objects of this flavor are actually created. This flavor is provided to make it easy to handle any floating-point exception. It is built on **sys:arithmetic-error**.

Message	*Value returned*
:operation	A symbol indicating the operation that caused the exception.
:operands	The list of operands to the operation.
:non-trap-result	The result that would have been returned if this trap had been disabled.
:saved-float-operation-status	The value of **sys:float-operation-status** at the time of the exception.

Proceed type	*Action*
:new-value	Takes one argument and uses this value as the result of the operation.

sys:float-divide-by-zero *Flavor*

A floating-point division by zero was attempted. This flavor is built on **sys:divide-by-zero** and **sys:floating-point-exception**.

sys:floating-exponent-overflow *Flavor*

Overflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception**. The **:function** message returns the function that got the overflow, if it is known, and **nil** if it is not known. The **:new-value** proceed type is provided with one argument, a floating-point number to use instead.

sys:floating-exponent-underflow *Flavor*

Underflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception**. The **:function** message returns the function that got the underflow, if it is known, and **nil** if it is not known. The **:use-zero** proceed type is provided with no arguments; a **0.0** is used instead.

sys:float-inexact-result *Flavor*

A floating-point result does not exactly represent the operation's result, due to the fixed precision of floating-point representation. Since most floating-point calculations are inexact, the inexact-result trap is disabled by default. This flavor is built on **sys:floating-point-exception**.

sys:float-invalid-operation *Flavor*

An invalid floating-point operation was attempted, such as dividing infinity by infinity. This flavor is built on **sys:floating-point-exception**.

sys:float-invalid-compare-operation *Flavor*

This is built on and is identical to **sys:float-invalid-operation**, except that it does not expect a numeric result. This flavor is raised for any arithmetic comparison (**<**, **>**, ≤, ≥, **=**, ≠) in which at least one of the operands is a NaN (IEEE not-a-number object).

For example:

```
(< (/ 0.0 0.0) 0.0)
```

signals **sys:float-invalid-compare-operation** if you "proceed" from the invalid division by zero operation.

sys:negative-sqrt *Flavor*

Computing the square root of a negative number was attempted. This flavor is built on **sys:float-invalid-operation**.

sys:float-divide-zero-by-zero *Flavor*

A floating-point division of zero by zero was attempted. This flavor is built on **sys:float-invalid-operation** and **sys:float-divide-by-zero**. Most programs handle not this condition itself, but rather one of the component condition flavors.

Miscellaneous System Errors Not Categorized by Base Flavor

This group includes the following errors:

- **deallocate-resource**
- **sys:end-of-file**
- **sys:stream-closed**
- **sys:wrong-stack-group-state**
- **sys:draw-off-end-of-screen**
- **sys:draw-on-unprepared-sheet**
- **sys:bitblt-destination-too-small**
- **sys:bitblt-array-fractional-word-width**
- **sys:write-in-read-only**
- **sys:pdl-overflow**
- **sys:area-overflow**
- **sys:virtual-memory-overflow**
- **sys:region-table-overflow**
- **sys:cons-in-fixed-area**

- **sys:throw-tag-not-seen**
- **sys:disk-error**
- **sys:redefinition**
- **si:resource-extra-deallocation**
- **si:resource-error**
- **si:resource-object-not-found**

sys:end-of-file *Flavor*

A function doing input from a stream attempted to read past the end-of-file. The **:stream** message returns the stream.

sys:stream-closed *Flavor*

An operation that required a stream to be open was attempted on a closed stream. **sys:stream-closed** accepts the following messages and has corresponding required init keywords:

:attempt	Returns a string briefly describing the attempted action on **zl-user:stream**, for example, "read from"
:stream	Returns the stream

Example:

```
(error 'sys:stream-closed :attempt "write to" :stream self)
```

sys:wrong-stack-group-state *Flavor*

A stack group was in the wrong state to be resumed. The **:stack-group** message returns the stack group.

sys:draw-off-end-of-screen *Flavor*

Drawing graphics past the edge of the screen was attempted.

sys:draw-on-unprepared-sheet *Flavor*

A drawing primitive (such as **sys:%draw-line**) was used on a screen array not inside a **tv:prepare-sheet** special form. The **:sheet** message returns the sheet (window) that should have been prepared.

sys:bitblt-destination-too-small *Flavor*

The destination array of a **bitblt** was too small.

sys:bitblt-array-fractional-word-width *Flavor*

An array passed to **bitblt** does not have a first dimension that is a multiple of 32 bits. The **:array** message returns the array.

sys:write-in-read-only *Flavor*

Writing into a read-only portion of memory was attempted. The **:address** message returns the address at which the write was attempted.

sys:pdl-overflow *Flavor*

A stack (pdl) overflowed. The **:pdl-name** message returns the name of the stack (a string, such as "regular" or "special"). The **:grow-pdl** proceed type is provided, with no arguments; it increases the size of the stack. This is based on **dbg:debugger-condition**, not on **error**.

sys:area-overflow *Flavor*

The maximum-size (**:size** argument to **make-area**) was exceeded.

sys:virtual-memory-overflow *Flavor*

An irrecoverable error that is signalled when you run out of virtual memory.

sys:region-table-overflow *Flavor*

An irrecoverable error that is signalled when you run out of regions.

sys:cons-in-fixed-area *Flavor*

Allocation of storage from a fixed area of memory was attempted.

Message	*Value returned*
:area	name of the area
:region	region number

sys:throw-tag-not-seen *Flavor*

throw or **zl:*throw** was called, but no matching **catch** or **zl:*catch** was found.

Message	*Value returned*
:tag	Catch-tag that was being thrown to.
:values	List of the values that were being thrown. If **zl:*throw** was called, this is always a list of two elements, the value being thrown and the tag; if the **throw** special form of Common Lisp is used, the list can be of any length.

The **:new-tag** proceed type is provided with one argument, a new tag (a symbol) to try instead of the original.

sys:redefinition *Flavor*

This is a simple condition rather than an error condition. It signals an attempt to redefine something by some other file than the one that originally defined it. The **:definition-type** argument specifies the kind of definition: it might be **defun** if the function cell is being defined, **zl:defstruct** if a structure is being defined, and so on.

Message	*Value returned*
:name	symbol (or function spec) being redefined
:old-pathname	pathname that originally defined it
:new-pathname	pathname that is now trying to define it

Either pathname is **nil** if the definition was from inside the Lisp environment rather than from loading a file.

The following proceed types are provided:

Message	*Action*
:proceed	Redefinition should go ahead; in the future no warnings should be signalled for this pair of pathnames.
:inhibit-definition	Definition is not changed and execution proceeds.
:no-action	Function should be redefined as if no warning had occurred.

Note: if this condition is not handled, the action is controlled by the value of **sys:inhibit-fdefine-warnings**.

Resource Errors Based on si:resource-error

This group includes the following errors:

- **si:resource-error**
- **si:resource-extra-deallocation**
- **si:resource-object-not-found**

si:resource-error *Flavor*

All resource-related error conditions are built on **si:resource-error**. Used primarily for **zl:typep**.

si:resource-object-not-found *Flavor*

Signifies an error in the client and gives the error message "Object not found in resource". This occurs when a deallocated object was not found in the resource.

This situation can be created in two ways:

- Not creating the object on the resource with the following:

```
(si:allocate-resource <resource name>...)
```

- Executing the following form between the original allocation, and the deallocation:

```
(si:clear-resource <resource name>)
```

Use the **:no-action** proceed type to ignore this error. The **:object** message returns the object. The **:resource** message returns the resource.

si:resource-extra-deallocation *Flavor*

Detects situations where there is extra deallocation, and enters the Debugger. Extra deallocation occurs when **deallocate-resource** is called more than one time on an object.

Use the **:no-action** message to ignore this error. The **:object** message returns the object. The **:resource** message returns the resource.

Function-Calling errors

This group includes the following errors:

- **sys:zero-args-to-select-method**
- **sys:too-few-arguments**
- **sys:too-many-arguments**
- **sys:wrong-type-argument**

sys:zero-args-to-select-method *Flavor*

A select method was applied to no arguments.

sys:too-few-arguments *Flavor*

A function was called with too few arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:additional-arguments** proceed type is provided with one argument, a list of additional argument values to which the function should be applied. If the error is proceeded, these new arguments are appended to the old arguments and the function is called with this new argument list.

sys:too-many-arguments *Flavor*

A function was called with too many arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:fewer-arguments** proceed type is provided with one argument, the new number of arguments with which the function should be called. In proceeding from this error, the function is called with the first *n* arguments only, where *n* is the number specified.

sys:wrong-type-argument *Flavor*

A function was called with at least one argument of invalid type.

Message	*Value returned*
:function	function with invalid argument(s)
:old-value	invalid value
:description	description of valid value
:arg-name	name of the argument
:arg-number	number of the argument (the first argument to a function is **0**, and so on) or **nil** if this does not apply

:description, **:arg-name**, and **:arg-number** are valid messages only when the error was signalled by **zl:check-arg**, **zl:check-arg-type**, or **zl:argument-typecase**. Check to be sure that the message is valid before sending it (remember **:operation-handled-p**).

Proceed type	*Action*
:argument-value	Takes one argument, the new value to use for the argument.
:store-argument-value	Takes one argument, the new value to use and to store back into the local variable in which it was found.

Array Errors

This group includes the following errors:

- **dbg:bad-array-mixin**
- **sys:bad-array-type**

- **sys:array-has-no-leader**
- **sys:array-wrong-number-of-dimensions**
- **sys:array-wrong-number-of-subscripts**
- **sys:number-array-not-allowed**
- **sys:subscript-out-of-bounds**

dbg:bad-array-mixin *Flavor*

Errors involving an array that seems to be the wrong object include this flavor. This condition flavor is never instantiated. It provides the **:array** message, which returns the array.

Proceed type	*Action*
:new-array	Takes one argument, an array to use instead of the old one.
:store-new-array	Takes one argument, an array to use instead of the old one and to store back into the local variable in which it was found.

sys:bad-array-type *Flavor*

A meaningless array type code was found in virtual memory, indicating a system bug. The **:type** message returns the numeric type code.

sys:array-has-no-leader *Flavor*

Using the leader of an array that has no array leader was attempted. The **:array** message returns the array. This includes the **dbg:bad-array-mixin** flavor.

sys:array-wrong-number-of-dimensions *Flavor*

The rank of the array provided was wrong; the array is in error and the subscripts are correct.

Message	*Value returned*
:dimensions-given	Number of subscripts presented.
:dimensions-expected	Number that should have been given.
:array	The array.

This includes the **dbg:bad-array-mixin** flavor.

sys:array-wrong-number-of-subscripts *Flavor*

This assumes that the array is correct and that the user/application caused the error by providing the incorrect number of subscripts.

Message	*Value returned*
:array	The array.
:subscripts-given	A list of the subscripts given.
:number-of-subscripts-given	The number of subscripts given.
:number-of-subscripts-expected	The rank of the array .

The following example signals **sys:array-wrong-number-of-subscripts**:

```
(array-in-bounds-p some-3-dimensional-array 2 3)
```

sys:number-array-not-allowed *Flavor*

A number array (such as an **sys:art-4b** or **sys:art-16b**) was used in a context in which number arrays are not valid, such as an attempt to make a pointer to an element with **zl:aloc** or **locf**. This includes the **dbg:bad-array-mixin** flavor.

sys:subscript-out-of-bounds *Flavor*

An attempt was made to reference an array using out-of-bounds subscripts, an out-of-bounds array leader element, or an out-of-bounds instance variable.

Message	*Value returned*
:object	The object (an array or instance) if it is known, and **nil** otherwise.
:function	Function that did the reference, or **nil** if it is not known.
:subscript-used	The subscript that was actually used.
:subscript-limit	The limit that it passed.

The individual subscripts are reported for the **:subscript-used** and **:subscript-limit** messages. These values are fixnums; if a multidimensional array was used, they are computed products.

Proceed type	*Action*
:new-subscript	Takes an arbitrary number of arguments, the new subscripts for the array reference.
:store-new-subscript	Takes the same arguments as **:new-subscript** and stores them back into the local variables in which they were found.

Eval Errors

This group includes the following errors:

- **sys:invalid-function**
- **sys:undefined-keyword-argument**
- **sys:unclaimed-message**

sys:invalid-function *Flavor*

The evaluator attempted to apply an object that is not a function or a symbol whose definition is an object that is not a function. The **:function** message returns the object that was applied. **sys:invalid-function** is signalled but is not proceedable.

sys:undefined-keyword-argument *Flavor*

The evaluator attempted to pass a keyword to a function that does not recognize that keyword.

Message	*Value returned*
:keyword	Unrecognized keyword.
:value	The value passed with it.

Proceed type	*Action*
:no-action	The keyword and its value are ignored.
:new-keyword	Specifies a new keyword to use instead. Its one argument is the new keyword.

sys:unclaimed-message *Flavor*

This flavor is built on **error**. The flavor system signals this error when it finds a message for which no method is available.

Message	*Value returned*
:object	The object.
:message	The message-name.
:arguments	The arguments of the message.

The object can be an instance or a select method.

Interning Errors Based on sys:package-error

This group includes the following errors:

- **sys:package-error**
- **sys:package-not-found**
- **sys:package-locked**

sys:package-error *Flavor*

All package-related error conditions are built on **sys:package-error**.

sys:package-not-found *Flavor*

A package-name lookup did not find any package by the specified name.

The **:name** message returns the name. The **:relative-to** message returns **nil** if only absolute names are being searched, or else the package whose relative names are also searched.

The **:no-action** proceed type can be used to try again. The **:new-name** proceed type can be used to specify a different name or package. The **:create-package** proceed type creates the package with default characteristics.

sys:package-locked *Flavor*

There was an attempt to intern a symbol in a locked package.

The **:symbol** message returns the symbol. The **:package** message returns the package.

The **:no-action** proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

Errors Involving Lisp Printed Representations

This group includes the following errors:

- **sys:print-not-readable**
- **sys:parse-error**
- **zl:parse-ferror**
- **sys:read-error**
- **sys:read-premature-end-of-symbol**
- **sys:read-end-of-file**
- **sys:read-list-end-of-file**
- **sys:read-string-end-of-file**

sys:print-not-readable *Flavor*

The Lisp printer encountered an object that it cannot print in a way that the Lisp reader can understand. The printer signals this condition only if **si:print-readably** is not **nil** (it is normally **nil**). The **:object** message returns the object. The **:no-action** proceed type is provided; proceeding this way causes the object to be printed as if **si:print-readably** were **nil**.

sys:parse-error *Flavor*

This flavor is built on **error** and is the type of error caught by the input editor. This flavor accepts the init keyword **:correct-input**. If the value is **t**, which is the default, the input editor prints "Type RUBOUT to correct your input" and does not erase the message until a non-self-inserting character is typed. If the value is **nil**,

no message is printed, and any typeout from the read function is erased immediately after the next character is typed. Syntax errors signalled by read functions should be built on top of this flavor.

zl:parse-ferror *Flavor*

This flavor is built on **sys:parse-error** and **zl:ferror**. It accepts the init keywords **:format-string** and **:format-args** as well as **:correct-input**. This flavor exists for read functions that do not have a special flavor of error defined for them.

sys:read-error *Flavor*

This flavor, built on **sys:parse-error**, includes errors encountered by the Lisp reader.

sys:read-premature-end-of-symbol *Flavor*

This is a new error flavor based on **sys:read-error**. It can be used for signalling when some read function finishes reading in the middle of a string that was supposed to contain a single expression.

Message	*Value returned*
:short-symbol	The symbol that was read.
:original-string	The string that it was reading from when it finished in the middle.

An example of the use of **sys:read-premature-end-of-symbol** is in **zwei:symbol-from-string**.

sys:read-end-of-file *Flavor*

The Lisp reader encountered an end-of-file while in the middle of a string or list. This flavor is built on **sys:read-error** and **sys:end-of-file**.

sys:read-list-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a list. This is built on **sys:read-end-of-file**. The **:list** message returns the list that was being built.

sys:read-string-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a string. This is built on **sys:read-end-of-file**. The **:string** message returns the string that was being built.

File-System Errors

The following condition flavors are part of Genera's generic file system interface. These flavors work for all file systems, whether local Lisp Machine File Systems (LMFS), remote LMFSes (accessed over a network), or remote file systems of other kinds, such as UNIX or TOPS-20. All of them report errors uniformly.

Some of these condition flavors describe situations that can occur during any file system operation. These include not only the most basic flavors, such as **fs:file-request-failure** and **fs:data-error**, but also flavors such as **fs:file-not-found** and **fs:directory-not-found**. Other file system condition flavors describe failures related to specific file system operations, such as **fs:rename-failure**, and **fs:delete-failure**. Given all these choices, you have to determine what condition is appropriate to handle, for example in checking for success of a rename operation. Would **fs:rename-failure** include cases where, say, the directory of the file being renamed is not found?

The answer to this question is that you should handle **fs:file-operation-failure**. **fs:rename-failure** and all other conditions at that level are signalled only for errors that relate specifically to the semantics of the operation involved. If you cannot delete a file because the file is not found, **fs:file-not-found** would be signalled. Suppose you cannot delete the file because its "don't delete switch" is set, which is an error relating specifically to deletion. **fs:delete-failure** would be signalled. Therefore, since you cannot know whether a condition flavor related to an operation requested or some more general error is signalled, you usually want to handle one of the most general flavors of file system error.

Under normal conditions, you would bind only for **fs:file-request-failure** or **fs:file-operation-failure** rather than for the more specific condition flavors described in this section. Some guidelines for using the different classes of errors:

error
: Any error at all. It is not wise in general to attempt to handle this, because it catches program and operating system bugs as well as file-related bugs, thus "hiding" knowledge of the system problems from you.

fs:file-error
: Any file related error at all. This includes **fs:file-operation-failure** as well as **fs:file-request-failure**. Condition objects of flavor **fs:file-request-failure** usually indicate that the file system, host operating system, or network did not operate properly. If your program is attempting to handle file-related errors, it should not handle these: it is usually better to allow the program to enter the debugger. Thus it is very rare that one would want to handle **fs:file-error**.

fs:file-operation-failure
: This includes almost all predictable file-related errors, whether they are related to the semantics of a specific operation, or are capable of occurring during many kinds of operations. Therefore, **fs:file-operation-failure** is usually the appropriate condition to handle.

Specific conditions — It is appropriate and correct to handle specific conditions, like **fs:delete-failure**, if your program assigns specific meaning to (or has specific actions associated with) specific occurrences, such as a nonexistent directory or an attempt to delete a protected file. If you do not "care" about specific conditions, but you wish to handle predictable file-related errors, you should handle **fs:file-operation-failure**. You should *not* attempt to handle, say, **fs:delete-failure** to test for any error occurring during deletion; it does not mean that.

File-system errors include the following major groups:

- **fs:file-error** Errors
- **fs:file-request-failure** Errors
- **fs:file-operation-failure** Errors
- Request Failures Based on **fs:file-request-failure**
- Login Errors
- File Lookup Errors
- **fs:access-error** Errors
- **fs:invalid-pathname-syntax** Errors
- **fs:wrong-kind-of-file** Errors
- **fs:creation-failure** Errors
- **fs:rename-failure** Errors
- **fs:change-property-failure** Errors
- **fs:delete-failure** Errors
- Errors Loading Binary Files
- Miscellaneous Operations Failure Errors

fs:file-error *Flavor*

This set includes errors encountered during file operations. This flavor is built on **error**.

Message	*Value returned*
:pathname	Pathname that was being operated on or **nil**.
:operation	Name of the operation that was being done: this is a keyword symbol such as **:open**, **:close**, **:delete**, or **:change-properties**, and it might be **nil** if the signaller does not know what the operation was or if no specific operation was in progress.

In a few cases, the **:retry-file-error** proceed type is provided, with no arguments; it retries the file system request. All flavors in this section accept these messages and might provide this proceed type.

fs:file-request-failure *Flavor*

This set includes all file-system errors in which the request did not manage to get to the file system.

fs:file-operation-failure *Flavor*

This set includes all file-system errors in which the request was delivered to the file system, and the file system decided that it was an error.

Note: every file-system error is either a request failure or an operation failure, and the rules given above explain the distinction. However, these rules are slightly unclear in some cases. If you want to be sure whether a certain error is a request failure or an operation failure, consult the detailed descriptions in the rest of this section.

Request Failures Based on fs:file-request-failure

This group includes the following errors:

- **fs:data-error**
- **fs:host-not-available**
- **fs:no-file-system**
- **fs:network-lossage**
- **fs:not-enough-resources**
- **fs:unknown-operation**

fs:data-error *Flavor*

Bad data is in the file system. This might mean data errors detected by hardware, or inconsistent data inside the file system. This flavor is built on **fs:file-request-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is provided in some cases; send a **:proceed-types** message to find out.

fs:host-not-available *Flavor*

The file server or file system is intentionally denying service to users. This does *not* mean that the network connection failed; it means that the file system explicitly does not care to be available. This flavor is built on **fs:file-request-failure**.

fs:no-file-system *Flavor*

The file system is not available. For example, this host does not have any file system, or this host's file system cannot be initialized for some reason. This flavor is built on **fs:file-request-failure**.

fs:network-lossage *Flavor*

The file server had some sort of trouble trying to create a new data connection and was unable to do so. This flavor is built on **fs:file-request-failure**.

fs:not-enough-resources *Flavor*

Some resource was not available in sufficient supply. Retrying the operation might work if you wait for some other users to go away or if you close some of your own files. This flavor is built on **fs:file-request-failure**.

fs:unknown-operation *Flavor*

An unsupported file-system operation was attempted. This flavor is built on **fs:file-request-failure**.

Login Errors

Some login problems are correctable, and some are not. To handle any correctable login problem, you set up a handler for **fs:login-required** rather than handling the individual conditions.

The correctable login problem conditions work in a special way. Genera's generic file system interface, in the user-end of the remote file protocol, always handles these errors with its own condition handler; it then signals the **fs:login-required** condition. Therefore, to handle one of these problems, you set up a handler for **fs:login-required**. The condition object for the correctable login problem can be obtained from the condition object for **fs:login-required**, by sending it an **:original-condition** message.

This group includes the following errors:

- **fs:login-problems**
- **fs:correctable-login-problems**
- **fs:unknown-user**
- **fs:invalid-password**
- **fs:not-logged-in**
- **fs:login-required**

fs:login-problems *Flavor*

This set includes all problems encountered while trying to log in to the file system. Currently, none of these ever happens when you use a local file system. This flavor is built on **fs:file-request-failure**.

fs:correctable-login-problems *Flavor*

This set includes all correctable problems encountered while trying to log in to the foreign host. None of these ever happens when you use a local file system. This flavor is built on **fs:login-problems**.

fs:unknown-user *Flavor*

The specified user name is unknown at this host. The **:user-id** message returns the user name that was used. This flavor is built on **fs:correctable-login-problems**.

fs:invalid-password *Flavor*

The specified password was invalid. This flavor is built on **fs:correctable-login-problems**.

fs:not-logged-in *Flavor*

A file operation was attempted before logging in. Normally the file system interface always logs in before doing any operation, but this problem can come up in certain unusual cases in which logging in has been aborted. This flavor is built on **fs:correctable-login-problems**.

fs:login-required *Flavor*

This is a simple condition built on **condition**. It is signalled by the file-system interface whenever one of the correctable login problems happens.

Message	*Value returned*
(send (send error :access-path) :host)	The foreign host.
:host-user-id	User name that would be the default for this host.
:original-condition	Condition object of the correctable login problem.

The **:password** proceed type is provided with two arguments, a new user name and a new password, both of which should be strings. If the condition is not handled by any handler, the file system prompts the user for a new user name and password, using the **zl:query-io** stream.

File Lookup

This group includes the following errors:

- **fs:file-lookup-error**
- **fs:file-not-found**
- **fs:multiple-file-not-found**
- **fs:directory-not-found**
- **fs:device-not-found**
- **fs:link-target-not-found**

fs:file-lookup-error *Flavor*

This set includes all file-name lookup errors. This flavor is built on **fs:file-operation-failure**.

fs:file-not-found *Flavor*

The file was not found in the containing directory. The TOPS-20 and TENEX "no such file type" and "no such file version" errors also signal this condition. This flavor is built on **fs:file-lookup-error**.

fs:multiple-file-not-found *Flavor*

None of a number of possible files was found. This flavor is built on **fs:file-lookup-error**. It is signalled when **load** is not given a specific file type but cannot find either a source or a binary file to load.

The flavor allows three init keywords of its own. These are also the names of messages that return the following:

:operation	The operation that failed.
:pathname	The pathname given to the operation.
:pathnames	A list of pathnames that were sought unsuccessfully.

The condition has a **:new-pathname** proceed type to prompt for a new pathname.

fs:directory-not-found *Flavor*

The directory of the file was not found, or does not exist. This means that the containing directory was not found. If you are trying to open a directory, and the actual directory you are trying to open is not found, **fs:file-not-found** is signalled. This flavor is built on **fs:file-lookup-error**.

This flavor has two Debugger special commands: **:create-directory**, to create only the lowest level of directory, and **:create-directories-recursively**, to create any missing superiors as well.

Errors of this flavor support the **:directory-pathname** message. This message, which can be sent to any such error, returns (when possible) a "pathname as directory" for the actual directory that was not found.

Example:

Assume the directory x:>a>b exists, but has no inferiors. The following produces an error instance to which **:pathname** produces #<LMFS-PATHNAME x:>a>b>c>d>thing.lisp> and **:directory-pathname** produces #<LMFS-PATHNAME x:>a>b>c> >.

```
(open "x:>a>b>c>d>thing.lisp")
```

Note: Not all hosts and access media can provide this information (currently, only local LMFS and LMFS accessed via NFILE can). When a host does not return this

information, **:directory-pathname** returns the same as **:pathname**, whose value is a pathname as directory for the best approximation known to the identity of the missing directory.

fs:device-not-found *Flavor*

The device of the file was not found, or does not exist. This flavor is built on **fs:file-lookup-error**.

fs:link-target-not-found *Flavor*

The target of the link that was opened did not exist. This flavor is built on **fs:file-lookup-error**.

fs:access-error Errors

This group includes the following errors:

- **fs:access-error**
- **fs:incorrect-access-to-file**
- **fs:incorrect-access-to-directory**

fs:access-error *Flavor*

This set includes all protection-violation errors. This flavor is built on **fs:file-operation-failure**.

fs:incorrect-access-to-file *Flavor*

Incorrect access to the file in the directory was attempted. This flavor is built on **fs:access-error**.

fs:incorrect-access-to-directory *Flavor*

Incorrect access to some directory containing the file was attempted. This flavor is built on **fs:access-error**.

fs:invalid-pathname-syntax Errors

This group includes the following errors:

- **fs:invalid-pathname-syntax**
- **fs:invalid-wildcard**
- **fs:wildcard-not-allowed**

fs:invalid-pathname-syntax *Flavor*

This set includes all invalid pathname syntax errors. This is not the same as **fs:parse-pathname-error**. These errors occur when a successfully parsed pathname object is given to the file system, but something is wrong with it. See the specific conditions that follow. This flavor is built on **fs:file-operation-failure**.

fs:invalid-wildcard *Flavor*

The pathname is not a valid wildcard pathname. This flavor is built on **fs:invalid-pathname-syntax**.

fs:wildcard-not-allowed *Flavor*

A wildcard pathname was presented in a context that does not allow wildcards. This flavor is built on **fs:invalid-pathname-syntax**.

fs:wrong-kind-of-file Errors

This group includes the following errors:

- **fs:wrong-kind-of-file**
- **fs:invalid-operation-for-link**
- **fs:invalid-operation-for-directory**

fs:wrong-kind-of-file *Flavor*

This set includes errors in which an invalid operation for a file, directory, or link was attempted.

fs:invalid-operation-for-link *Flavor*

The specified operation is invalid for links, and this pathname is the name of a link. This flavor is built on **fs:wrong-kind-of-file**.

fs:invalid-operation-for-directory *Flavor*

The specified operation is invalid for directories, and this pathname is the name of a directory. This flavor is built on **fs:wrong-kind-of-file**.

fs:creation-failure Errors

This group includes the following errors:

- **fs:creation-failure**
- **fs:file-already-exists**

- **fs:create-directory-failure**
- **fs:directory-already-exists**
- **fs:create-link-failure**

fs:creation-failure *Flavor*

This set includes errors related to attempts to create a file, directory, or link. This flavor is built on **fs:file-operation-failure**.

fs:file-already-exists *Flavor*

A file of this name already exists. This flavor is built on **fs:creation-failure**.

fs:create-directory-failure *Flavor*

This set includes errors related to attempts to create a directory. This flavor is built on **fs:creation-failure**.

fs:directory-already-exists *Flavor*

A directory or file of this name already exists. This flavor is built on **fs:create-directory-failure**.

fs:create-link-failure *Flavor*

This set includes errors related to attempts to create a link. This flavor is built on **fs:creation-failure**.

fs:rename-failure Errors

This group includes the following errors:

- **fs:rename-failure**
- **fs:rename-to-existing-file**
- **fs:rename-across-directories**
- **fs:rename-across-hosts**

fs:rename-failure *Flavor*

This set includes errors related to attempts to rename a file. The **:new-pathname** message returns the target pathname of the rename operation. This flavor is built on **fs:file-operation-failure**.

fs:rename-to-existing-file *Flavor*

The target name of a rename operation is the name of a file that already exists. This flavor is built on **fs:rename-failure**.

fs:rename-across-directories *Flavor*

The devices or directories of the initial and target pathnames are not the same, but on this file system they are required to be. This flavor is built on **fs:rename-failure**.

fs:rename-across-hosts *Flavor*

The hosts of the initial and target pathnames are not the same. This flavor is built on **fs:rename-failure**.

fs:change-property-failure Errors

This group includes the following errors:

- **fs:change-property-failure**
- **fs:unknown-property**
- **fs:invalid-property-value**

fs:change-property-failure *Flavor*

This set includes errors related to attempts to change properties of a file. This might mean that you tried to set a property that only the file system is allowed to set. For file systems without user-defined properties, it might mean that no such property exists. This flavor is built on **fs:file-operation-failure**.

fs:unknown-property *Flavor*

The property is unknown. This flavor is built on **fs:change-property-failure**.

fs:invalid-property-value *Flavor*

The new value provided for the property is invalid. This flavor is built on **fs:change-property-failure**.

fs:delete-failure Errors

This group includes the following errors:

- **fs:delete-failure**
- **fs:directory-not-empty**
- **fs:dont-delete-flag-set**

fs:delete-failure *Flavor*

This set includes errors related to attempts to delete a file. It applies to cases where the file server reports that it cannot delete a file. The exact events involved depend on what the host file server has received from the host. This flavor is built on **fs:file-operation-failure**.

fs:directory-not-empty *Flavor*

An invalid deletion of a nonempty directory was attempted. This flavor is built on **fs:delete-failure**.

fs:dont-delete-flag-set *Flavor*

Deleting a file with a "don't delete" flag was attempted. This flavor is built on **fs:delete-failure**.

Miscellaneous Operations Failures

This group includes the following errors:

- **fs:circular-link**
- **fs:unimplemented-option**
- **fs:inconsistent-options**
- **fs:invalid-byte-size**
- **fs:no-more-room**
- **fs:filepos-out-of-range**
- **fs:file-locked**
- **fs:file-open-for-output**
- **fs:not-available**

fs:circular-link *Flavor*

The pathname is a link that eventually gets linked back to itself. This flavor is built on **fs:file-operation-failure**.

fs:unimplemented-option *Flavor*

This set includes errors in which an option to a command is not implemented. This flavor is built on **fs:file-operation-failure**.

fs:inconsistent-options *Flavor*

Some of the options given in this operation are inconsistent with others. This flavor is built on **fs:file-operation-failure**.

fs:invalid-byte-size *Flavor*

The value of the "byte size" option was not valid. This flavor is built on **fs:unimplemented-option**.

fs:no-more-room *Flavor*

The file system is out of room. This can mean any of several things:

- The entire file system might be full.
- The particular volume that you are using might be full.
- Your directory might be full.
- You might have run out of your allocated quota.
- Other system-dependent things.

This flavor is built on **fs:file-operation-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is sometimes provided.

fs:filepos-out-of-range *Flavor*

Setting the file pointer past the end-of-file position or to a negative position was attempted. This flavor is built on **fs:file-operation-failure**.

fs:file-locked *Flavor*

The file is locked. It cannot be accessed, possibly because it is in use by some other process. Different file systems can have this problem in various system-dependent ways. This flavor is built on **fs:file-operation-failure**.

fs:file-open-for-output *Flavor*

Opening a file that was already opened for output was attempted. This flavor is built on **fs:file-operation-failure**.

Note: ITS, TOPS-20, and TENEX file servers do not use this condition; they signal **fs:file-locked** instead.

fs:not-available *Flavor*

The file or device exists but is not available. Typically, the disk pack is not mounted on a drive, the drive is broken, or the like. Probably, operator intervention is required to fix the problem: retrying the operation is likely to work after the problem is solved. This flavor is built on **fs:file-operation-failure**. Do not confuse this with **fs:host-not-available**.

Errors Loading Binary Files

This group includes the following errors:

- **sys:binary-file-obsolete-version**
- **sys:binary-file-obsolete-version-3**

sys:binary-file-obsolete-version *Flavor*

A condition based on **sys:binary-file-obsolete-version** is signalled whenever the system reads a binary file whose version is obsolete but that can still be loaded, with incorrect results. If you want the file loaded, proceed from this condition with **:no-action**.

sys:binary-file-obsolete-version-3 *Flavor*

A condition based on **sys:binary-file-obsolete-version-3** is signalled whenever the system reads a version 3 (Release 6) binary file. The file can still be loaded, possibly with incorrect results. If you want the file loaded, proceed from this condition with **:no-action**.

Pathname Errors

Pathname errors include five major groups:

- **fs:pathname-error**
- **fs:parse-pathname-error**
- **fs:invalid-pathname-component**
- **fs:unknown-pathname-host**
- **fs:undefined-logical-pathname-translation**

fs:pathname-error *Flavor*

This set includes errors related to pathnames. This is built on the **error** flavor. The following flavors are built on this one.

fs:parse-pathname-error *Flavor*

A problem occurred in attempting to parse a pathname.

fs:invalid-pathname-component *Flavor*

Attempt to create a pathname with an invalid component.

Message	*Value returned*
:pathname	The pathname.
:component-value	The invalid value.
:component	The name of the component (a keyword symbol such as **:name** or **:directory**).
:component-description	A "pretty name" for the component (such as "file name" or "directory").

The **:new-component** proceed type is defined with one argument, a component value to use instead.

At the time this is signalled, a pathname object with the invalid component has actually been created; this is what the **:pathname** message returns. The error is signalled just after the pathname object is created, before it goes in the pathname hash table.

fs:unknown-pathname-host *Flavor*

The function **fs:get-pathname-host** was given a name that is not the name of any known file computer. The **:name** message returns the name (a string).

fs:undefined-logical-pathname-translation *Flavor*

A logical pathname was referenced but is not defined. The **:logical-pathname** message returns the logical pathname. This flavor has a **:define-directory** proceed type, which prompts for a physical pathname whose directory component is the translation of the logical directory on the given host.

Network Errors

Network errors include four major groups:

- **sys:network-error**
- Local Network Problems
- Remote Network Problems
- Connection Problems

sys:network-error *Flavor*

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

Local Network Errors

This group includes the following errors:

- **sys:local-network-error**
- **sys:network-resources-exhausted**
- **sys:unknown-address**
- **sys:unknown-host-name**

sys:local-network-error *Flavor*

This set includes network errors related to problems with your own Symbolics computer, rather than with the network or the foreign host. This flavor is built on **sys:network-error**.

sys:network-resources-exhausted *Flavor*

The local network control program exhausted some resource; for example, its connection table is full. This flavor is built on **sys:local-network-error**.

sys:unknown-address *Flavor*

The network control program was given an address that is not understood. The **:address** message returns the address. This flavor is built on **sys:local-network-error**.

sys:unknown-host-name *Flavor*

The host parser (**net:parse-host**) was given a name that is not the name of any known host. The **:name** message returns the name. This flavor is built on **sys:local-network-error**.

Remote Network Errors

This group includes the following errors:

- **sys:remote-network-error**
- **sys:bad-connection-state**
- **sys:connection-error**
- **sys:host-not-responding**

sys:remote-network-error *Flavor*

This set includes network errors related to problems with the network or the foreign host, rather than with your Symbolics computer.

Message	*Value returned*
:foreign-host	The remote host.
:connection	The connection, or **nil** if no particular connection is involved.

This flavor is built on **sys:network-error**.

sys:bad-connection-state *Flavor*

This set includes remote errors in which a connection enters a bad state. This flavor is built on **sys:remote-network-error**. It actually can happen due to local causes, however. In particular, if your Symbolics computer stays inside a **without-interrupts** for a long time, the network control program might decide that a host is not answering periodic status requests and put its connections into a closed state.

sys:connection-error *Flavor*

This set includes remote errors that occur while trying to establish a new network connection. The **:contact-name** message to any error object in this set returns the contact name that you were trying to connect to. This flavor is built on **sys:remote-network-error**.

sys:host-not-responding *Flavor*

This set includes errors in which the host is not responding, whether during initial connection or in the middle of a connection. This flavor is built on **sys:remote-network-error**.

Connection Errors

This group includes the following errors:

- **sys:host-not-responding-during-connection**
- **sys:host-stopped-responding**
- **sys:connection-refused**
- **sys:connection-closed**
- **sys:connection-closed-locally**
- **sys:connection-lost**
- **sys:connection-no-more-data**
- **sys:network-stream-closed**

sys:host-not-responding-during-connection *Flavor*

The network control program timed out while trying to establish a new connection to a host. The host might be down, or the network might be down. This flavor is built on **sys:host-not-responding** and **sys:connection-error**.

sys:host-stopped-responding *Flavor*

A host stopped responding during an established network connection. The host or the network might have crashed. This flavor is built on **sys:host-not-responding** and **sys:bad-connection-state**.

sys:connection-refused *Flavor*

The foreign host explicitly refused to accept the connection. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:connection-error**.

sys:connection-closed *Flavor*

An established connection was closed. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state**.

sys:connection-closed-locally *Flavor*

The local host closed the connection and then tried to use it. This flavor is built on **sys:bad-connection-state**.

sys:connection-lost *Flavor*

The foreign host reported a problem with an established connection and that connection can no longer be used. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state**.

sys:connection-no-more-data *Flavor*

No more data remains on this connection. This flavor is built on **sys:bad-connection-state**.

sys:network-stream-closed *Flavor*

This is a combination of **sys:network-error** and **sys:stream-closed** and is usually used as a base flavor by network implementations (for example, Chaos and TCP).

Tape Errors

Tape errors include four major groups:

- **tape:tape-error**
- **tape:mount-error**
- **tape:tape-device-error**
- **tape:end-of-tape**

tape:tape-error *Flavor*

This set includes all tape errors. This flavor is built on **error**.

tape:mount-error *Flavor*

A set of errors signalled because a tape could not be mounted. This includes problems such as "no ring" and "drive not ready". Normally, **tape:make-stream** handles these errors, and manages mount retry. This flavor is built on **tape:tape-error**.

tape:tape-device-error *Flavor*

A hardware data error, such as a parity error, controller error, or interface error, occurred. This flavor has **tape:tape-error** as a **:required-flavor**.

tape:end-of-tape *Flavor*

The end of the tape was encountered. When this happens on writing, the tape usually has a few more feet left, in which the program is expected to finish up and write two end-of-file marks. Normally, closing the stream does this automatically. Whether or not this error is ever seen on input depends on the tape controller. Most systems do not see the end of tape on reading, and rely on the software that wrote the tape to have cleanly terminated its data, with EOFs.

This flavor is built on **tape:tape-device-error** and **tape:tape-error**.

Condition Functions in the Common Lisp Package with Symbolics Common Lisp Extensions

Here is a list of condition functions in the *Conditions* chapter that have Symbolics Common Lisp extensions:

Function	*Extension(s)*
warn	*optional-options*, *optional-condition-name*
error	In Symbolics Common Lisp you can use **error** with the arguments *condition* and *init-options*, or with the argument *condition-object*.
cerror	*optional-condition-name*

Packages

The Need for Packages

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

Genera is a huge Lisp environment, in which many programs must coexist. All of the "operating system", the compiler, the editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program that you use during your session must be loaded into the same environment. Each of these programs is composed of a group of functions; each function must have its own distinct name to avoid conflicts. For example, if the compiler had a function named **pull**, and you loaded a program that had its own function named **pull**, the compiler's **pull** would be redefined, probably breaking the compiler.

It would not really be possible to prevent these conflicts, since the programs are written by many different people who could never get together to hash out who gets the privilege of using a specific name such as **pull**.

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

A *package* is a mapping from names to symbols. When two programs are not closely related and hence are likely to have conflicts over the use of names, the programs can use separate packages to enable each program to have a different mapping from names to symbols. In the example above, the compiler can use a package that maps the name **pull** into a symbol whose function definition is the compiler's **pull** function. Your program can use a different package that maps the name **pull** into a different symbol whose function definition is your function. When your program is loaded, the compiler's **pull** function is not redefined, because it is attached to a symbol that is not affected by your program. The compiler does not break.

The word "package" is used to refer to a mapping from names to symbols because a number of related symbols are packaged together into a single entity. Since the substance of a program (such as its function definitions and variables) consists of attributes of symbols, a package also packages together the parts of a program. The package system allows the author of a group of closely related programs that should share the same symbols to define a single package for those programs.

It is important to understand the distinction between a name and a symbol. A name is a sequence of characters that appears on paper (or on a screen or in a file). This is often called a *printed representation*. A symbol is a Lisp object inside the machine. You should keep in mind how Lisp reading and loading work. When a source file is read into Genera, or a compiled binary file is loaded in, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it uses a package to map that printed representation (a name) into the symbol itself. The loader does the same thing. The package system arranges to use the correct pack-

age whenever a file is read or loaded. (For a detailed explanation of this process: See the section "Specifying Packages in Programs".

Example of the Need for Packages

Suppose there are two programs named **chaos** and **arpa**, for handling the Chaosnet and Arpanet respectively. The author of each program wants to have a function called **get-packet**, which reads in a packet from the network. Also, each wants to have a function called **allocate-pbuf**, which allocates the packet buffer. Each "get" routine first allocates a packet buffer, and then reads bits into the buffer; therefore, each version of **get-packet** should call the respective version of **allocate-pbuf**.

Without the package system, the two programs could not coexist in the same Lisp environment. But the package system can be used to provide a separate space of names for each program. What is required is to define a package named **chaos** to contain the Chaosnet program, and another package **arpa** to hold the Arpanet program. When the Chaosnet program is read into the machine, the names it uses are translated into symbols via the **chaos** package. So when the Chaosnet program's **get-packet** refers to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** package is found, which is the **allocate-pbuf** of the Chaosnet program — the right one. Similarly, the Arpanet program's **get-packet** would be read in using the **arpa** package and would refer to the Arpanet program's **allocate-pbuf**.

Sharing of Symbols Among Packages

How the Package System Allows Symbol Sharing

Besides keeping programs isolated by giving each program its own set of symbols, the package system must also provide controlled sharing of symbols among packages. It would not be adequate for each package's set of symbols to be completely disjoint from the symbols of every other package. For example, almost every package ought to include the whole Lisp language: **car**, **cdr**, **format**, and so on should be available to every program.

There is a critical tension between these two goals of the package system. On the one hand, we want to keep the packages isolated, to avoid the need to think about conflicts between programs when we choose names for things. On the other hand, we want to provide connections among packages so that the facilities of one program can be made available to other programs. All the complexity of the package system arises from this tension. Almost all of the package system's features exist to provide easy ways to control the sharing of symbols among packages, while avoiding accidental unwanted sharing of symbols. Unexpected sharing of a symbol between packages, when the authors of the programs in those packages expected to have private symbols of their own, is a *name conflict* and can cause programs to go awry. See the section "Package Name-Conflict Errors".

Note that sharing symbols is not as simple as merely making the symbols defined by the Lisp language available in every package. A very important feature of Genera is *shared programs*; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with many other systems, where many different programs have been written to accomplish the same thing.

For example, the routines to manipulate a robot arm might be a separate program, residing in its own package. A second program called **blocks** (the blocks world, of course) wants to manipulate the arm, so it would want to call functions from the arm package. This means that the blocks program must have a way to name those robot arm functions. One way to do this is to arrange for the name-to-symbol mapping of the blocks package to map the names of those functions into the same identical symbols as the name-to-symbol mapping of the arm package. These symbols would then be shared between the two packages.

This sharing must be done with great care. The symbols to be shared between the two packages constitute an interface between two modules. The names to be shared must be agreed upon by the authors of both programs, or at least known to them. They cannot simply make every symbol in the arm program available to the blocks program. Instead they must define some subset of the symbols used by the arm program as its *interface* and make only those symbols available. Typically each name in the interface is carefully chosen (more carefully than names that are only used internally). The arm program comes with documentation listing the symbols that constitute its interface and describing what each is used for. This tells the author of the blocks program not only that a particular symbol is being used as the name of a function in the arms program (and thus cannot be used for a function elsewhere), but also what that function does (move the arm, for instance) when it is called.

The package system provides for several styles of interface between modules. For several examples of how the blocks program and the arm program might communicate, see the section "Examples of Symbol Sharing Among Packages".

An important aspect of the package system is that it makes it necessary to clarify the modularity of programs and the interfaces between them. The package system provides some tools to allow the interface to be explicitly defined and to check that everyone agrees on the interface.

External Symbols

The name-to-symbol mappings of a package are divided into two classes, *external* and *internal*. We refer to the symbols accessible via these mappings as being *external* and *internal* symbols of the package in question, though really it is the mappings that are different and not the symbols themselves. Within a given package, a name refers to one symbol or to none; if it does refer to a symbol, that symbol is either external or internal in that package, but not both.

External symbols are part of the package's public interface to other packages. These are supposed to be chosen with some care and are advertised to outside

users of the package. Internal symbols are for internal use only, and these symbols are normally hidden from other packages. Most symbols are created as internal symbols; they become external only if they are explicitly *exported* from a package.

A symbol can appear in many packages. It can be external in one package and internal in another. It is valid for a symbol to be internal in more than one package, and for a symbol to be external in more than one package. A name can refer to different symbols in different packages. However, a symbol always has the same name no matter where it appears. This restriction is imposed both for conceptual simplicity and for ease of implementation.

Package Inheritance

Some name-to-symbol mappings are established by the package itself, while others are inherited from other packages. When package A inherits mappings from package B, package A is said to *use* package B. A symbol is said to be *accessible* in a package if its name maps to it in that package, whether directly or by inheritance. A symbol is said to be *present* in a package if its name maps to it directly (not by inheritance). If a symbol is accessible to a package, then it can be referenced by a program that is read into that package. Inheritance allows a package to be built up by combining symbols from a number of other packages.

Package inheritance interacts with the distinction between internal and external symbols. When one package uses another, it inherits only the external symbols of that package. This is necessary in order to provide a well-defined interface and avoid accidental name conflicts. The external symbols are the ones that are carefully chosen and advertised. If internal symbols were inherited, it would be hard to predict just which symbols were shared between packages.

A package can use any number of other packages; it inherits the external symbols of all of them. If two of these external symbols had the same name it would be unpredictable which one would be inherited, so this is considered to be a name-conflict error. Consequently the order of the used packages is immaterial and does not affect what symbols are accessible.

Only symbols that are present in a package can be external symbols of that package. However, the package system hides this restriction by copying an inherited mapping directly into a package if you request that the symbol be exported. Note: When package A uses package B, it inherits the external symbols of B. But these do not become external symbols of A, and are not inherited by package C that uses package A. A symbol becomes an external symbol of A only by an explicit request to export it from A.

A package can be made to use another package by the **:use** option to **defpackage** or **make-package** or by calling the **use-package** function.

Global Packages

Almost every package should have the basic symbols of the Lisp language accessible to it. This includes:

- Symbols that are names of useful functions, such as **cdr**, **cons**, and **print**
- Symbols that are names of special forms, such as **cond**
- Symbols that are names of useful variables, such as ***read-base***, ***standard-output***, and ***
- Symbols that are names of useful constants, such as **lambda-list-keywords**
- Symbols that are used by the language as symbols in their own right, such as **&optional**, **t**, **nil**, and **special**

Rather than providing an explicit interface between every program and the Lisp language, listing explicitly the particular symbols from the Lisp language that that program intends to use, it is more convenient to make all the Lisp symbols accessible. Unless otherwise specified, every package inherits from a global package. Common Lisp packages inherit from **common-lisp-global** (or **cl**) and Zetalisp packages inherit from **global** (or **zl**). The external symbols of **common-lisp-global** are all the symbols of the Lisp language, including all the symbols documented without a colon (**:**) in their name. The **common-lisp-global** package has no internal symbols.

All programs share the global symbols, and cannot use them for private purposes. For example, the symbol **delete** is the name of a Lisp function and thus is in the **common-lisp-global** package. Even if a program does not use the **delete** function, it inherits the global symbol named **delete** and therefore cannot define its own function with that name to do something different. Furthermore, if two programs each want to use the symbol **delete** as a property list indicator, they can bump into each other because they do not have private symbols. You can use a mechanism called *shadowing* to declare that a private symbol is desired rather than inheriting the global symbol. See the section "Shadowing Symbols". You can also use the **where-is** function and the Where Is Symbol (m-X) editor command to determine whether a symbol is private or shared when writing a program.

Similar to the **common-lisp-global** package is the **system** package, which contains all the symbols that are part of the "operating system" interface or the machine architecture, but not regarded as part of the Lisp language. The **system** package is not inherited unless specifically requested.

Here is how package inheritance works in the example of the two network programs. (See the section "Example of the Need for Packages".) When the Chaosnet program is read into the Lisp world, the current package is the **chaos** package. Thus all of the names in the Chaosnet program are mapped into symbols by the **chaos** package. If there is a reference to some well-known global symbol such as **append**, it is found by inheritance from the **common-lisp-global** package, assuming no symbol by that name is present in the **chaos** package. If, however, there is a reference to a symbol that you created, a new symbol is created in the **chaos** package. Suppose the name **get-packet** is referenced for the first time. No symbol by this name is directly present in the **chaos** package, nor is such a symbol inher-

ited from **common-lisp-global**. Therefore the reader (actually the **intern** function) creates a new symbol named **get-packet** and makes it present in the **chaos** package. When **get-packet** is referred to later in the Chaosnet program, that symbol is found.

When the Arpanet program is read in, the current package is **arpa** instead of **chaos**. When the Arpanet program refers to **append**, it gets the **common-lisp-global** one; that is, it shares the same symbol that the Chaosnet program got. However, if it refers to **get-packet**, it does *not* get the same symbol the Chaosnet program got, because the **chaos** package is not being searched. Rather, the **arpa** and **common-lisp-global** packages are searched. A new symbol named **get-packet** is created and made present in the **arpa** package.

So what has happened is that there are two **get-packet**s: one for **chaos** and one for **arpa**. The two programs are loaded together without name conflicts.

Home Package of a Symbol

Every symbol has a *home package*. When a new symbol is created by the reader and made present in the current package, its home package is set to the current package. The home package of a symbol can be obtained with the **symbol-package** function.

Most symbols are present only in their home package; however, it is possible to make a symbol be present in any number of packages. Only one of those packages can be distinguished as the home package; normally this is the first package in which the symbol was present. The package system tries to ensure that a symbol *is* present in its home package. When a symbol is first created by the reader (actually by the **intern** function), it is guaranteed to be present in its home package. If the symbol is removed from its home package (by the **unintern** function), the home package of the symbol is set to **nil**, even if the symbol is still present in some other package.

Some symbols are not present in any package; they are said to be *uninterned*. See the section "Mapping Names to Symbols". The **make-symbol** function can be used to create such a symbol. An uninterned symbol has no home package; the **symbol-package** function returns **nil** given such a symbol.

When a symbol is printed, for example, with **prin1**, the printer produces a printed representation that the reader turns back into the same symbol. If the symbol is not accessible to the current package, a qualified name is printed. See the section "Qualified Package Names". The symbol's home package is used as the prefix in the qualified name.

Importing and Exporting Symbols

A symbol can be made accessible to packages other than its home package in two ways, *importing* and *exporting*.

Any symbol can be made present in a package by *importing* it into that package. This is how a symbol can be present in more than one package at the same time.

After importing a symbol into the current package, it can be referred to directly with an unqualified name. Importing a symbol does not change its home package, and does not change its status in any other packages in which it is present.

When a symbol is imported, if another symbol with the same name is already accessible to the package, a name-conflict error is signalled. The *shadowing-import* operation is a combination of shadowing (See the section "Shadowing Symbols") and importing; it resolves a name conflict by getting rid of any existing symbol accessible to the package.

Any number of symbols can be *exported* from a package. This declares them to be *external* symbols of that package and makes them accessible in any other packages that *use* the first package. To use a package means to inherit its external symbols.

When a symbol is exported, the package system makes sure that no name conflict is caused in any of the packages that inherit the newly exported symbol.

A symbol can be imported by using the **:import**, **:import-from**, or **:shadowing-import** option to **defpackage** and **make-package**, or by calling the **import** or **shadowing-import** function. A symbol can be exported by using the **:export** option to **defpackage** or **make-package**, or by calling the **export** function. See the section "Defining a Package". See the section "Functions that Import, Export, and Shadow Symbols".

Shadowing Symbols

You can avoid inheriting unwanted symbols by *shadowing* them. To shadow a symbol that would otherwise be inherited, you create a new symbol with the same name and make it present in the package. The new symbol is put on the package's list of shadowing symbols, to tell the package system that it is not an accident that there are two symbols with the same name. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. Shadowing allows the creator of a package to avoid name conflicts that are anticipated in advance.

As an example of shadowing, suppose you want to define a function named **nth** that is different from the normal **nth** function. (Perhaps you want **nth** to be compatible with the Interlisp function of that name.) Simply writing **(defun nth ...)** in your program would redefine the system-provided **nth** function, probably breaking other programs that use it. (The system detects this and queries you before proceeding with the redefinition.)

The way to resolve this conflict is to put the program (call it snail) that needs the incompatible definition of **nth** in its own package and to make the **snail** package shadow the symbol **nth**.

Now there are two symbols named **nth**, so defining **snail**'s **nth** to be an Interlisp-compatible function does not affect the definition of the global **nth**. Inside the snail program, the global symbol **nth** cannot be seen, which is why we say that it is shadowed. If some reason arises to refer to the global symbol **nth** inside the snail program, the qualified name **common-lisp-global:nth** can be used.

A shadowing symbol can be established by the **:shadow** or **:shadowing-import** option to **defpackage** or **make-package**, or by calling the **shadow** or **shadowing-import** function. See the section "Functions that Import, Export, and Shadow Symbols".

The Keyword Package

The Lisp reader is not context-sensitive; it reads the same printed representation as the same symbol regardless of whether the symbol is being used as the name of a function, the name of a variable, a quoted constant, a syntactic word in a special form, or anything else. The consistency and simplicity afforded by this lack of context sensitivity are very important to Lisp's interchangeability of programs and data, but they do cause a problem in connection with packages. If a certain function is to be shared between two packages, then the symbol that names that function has to be shared for all contexts, not just for functional context. This can accidentally cause a variable, or a property list indicator, or some other use of a symbol, to be shared between two packages when not desired. Consequently, it is important to minimize the number of symbols that are shared between packages, since every such symbol becomes a "reserved word" that cannot be used without thinking about the implications. Furthermore, the set of symbols shared among all the packages in the world is not legitimately user-extensible, because adding a new shared symbol could cause a name conflict between unrelated programs that use symbols by that name for their own private purposes.

On the other hand, there are many important applications for which the package system just gets in the way and one would really like to have *all* symbols shared between packages. Typically this occurs when symbols are used as objects in their own right, rather than just as names for things.

This dilemma is partially resolved by the introduction of *keywords* into the language. Keywords are a set of symbols that is disjoint from all other symbols and exist as a completely independent set of names. There is no separation of packages as far as keywords are concerned; all keywords are available to all packages and two distinct keywords cannot have the same name. Of course, a keyword can have the same name as one or more ordinary symbols. To distinguish keywords from ordinary symbols, the printed representation of a keyword starts with a colon (**:**) character. The sharing of keywords among all packages does not affect the separation of ordinary symbols into private symbols of each package.

Specifying Packages in Programs

If you are an inexperienced user, you need never be aware of the existence of packages when writing programs. The **user** package is selected by default as the package for reading expressions typed at the Lisp Listener. Files are read in the **user** package if no package is specified. Since all the functions that users are likely to need are provided in the **global** package, which is used by **user**, they are all accessible. In the documentation, functions that are not in the **global** package are documented with colons in their names, so typing the name the way it is docu-

mented works. Keywords, of course, must be typed with a prefix colon, but since that is the way they are documented it is possible to regard the colon as just part of the name, not as anything having to do with packages.

The current package is the value of the variable ***package*** (**zl:package**). The current package in the "selected" process is displayed in the status line. This allows you to tell how forms you type in are read.

If you are writing a program that you expect others to use, you should put it in some package other than **user**, so that its internal functions do not conflict with names other users use. For whatever reason, if you are loading your programs into packages other than **user**, you need to know about special constructs including **defpackage**, qualified names, and file attribute lists. See the section "Defining a Package". See the section "Qualified Package Names".

Obviously, every file must be loaded into the right package to serve its purpose. It might not be so obvious that every file must be compiled in the right package, but it is just as true. Any time the names of symbols appearing in the file must be converted to the actual symbols, the conversion must take place relative to a package.

The system usually decides which package to use for a file by looking at the file's *attribute list*. See the section "File Attribute Lists". A compiled file remembers the name of the package it was compiled in, and loads into the same package. In the absence of any of these specifications, the package defaults to the current value of ***package***, which is usually the **user** package unless you change it.

The file attribute list of a character file is the line at the front of the file that looks something like:

```
;;; -*- Mode:Lisp; Package:System-Internals -*-
```

This specifies that the package whose name or nickname is **system-internals** is to be used. Alphabetic case does not matter in these specifications. Relative package names are not used, since there is no meaningful package to which the name could be relative. See the section "Relative Package Names".

If the package attribute contains parentheses, then the package is automatically created if it is not found. This is useful when a single file is in its own package, not shared with any other files, and no special options are required to set up that package. The valid forms of package attribute are:

-*- Package: *Name* **-*-**
: Signal an error if the package is not found, allowing you to load the package's definition from another file, specify the name of an existing package to use instead, or create the package with default characteristics.

-*- Package: (*Name***) -*-**
: If the package is not found, create it with the specified name and default characteristics. It uses **global** so that it inherits the Lisp language symbols.

-*- Package: (*Name use***) -*-**
: If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages.

-*- Package: (*Name use size***) -*-**

If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages. *size* is a decimal number, the number of symbols that expected to be present in the package.

-*- Package: (*Name keyword value keyword value***...) -*-**

If the package is not found, create it with the specified name. The rest of the list supplies the keyword arguments to **make-package**. In the event of an ambiguity between this form and the previous one, the previous one is preferred. You can avoid ambiguity by specifying more than one keyword.

Binary files have similar file attribute lists. The compiler always puts in a **:package** attribute to cause the binary file to be loaded into the same package it was compiled in, unless this attribute is overridden by arguments to **load**.

Package Names

Introduction to Package Names

Each package has a name and perhaps some nicknames. These are assigned when the package is created, though they can be changed later. A package's name should be something long and self-explanatory like **editor**; there might be a nickname that is shorter and easier to type, like **ed**. Typically the name of a package is also the name of the program that resides in that package.

There is a single namespace for packages. Instead of setting up a second-level package system to isolate names of packages from each other, we simply say that package name conflicts are to be resolved by using long explanatory names. There are sufficiently few packages in the world that a mechanism to allow two packages to have the same name does not seem necessary. Note that for the most frequent use of package names, qualified names of symbols, name clashes between packages can be alleviated using relative names.

The syntax conventions for package names are the same as for symbols. When the reader sees a package name (as part of a qualified symbol name), alphabetic characters in the package name are converted to uppercase unless preceded by the "/" escape character or unless the package name is surrounded by "|" characters. When a package name is printed by the printer, if it does not consist of all uppercase alphabetics and non-delimiter characters, the "/" and "|" escape characters are used.

Package name lookup is case-insensitive, but it it is not considered good style to have two packages whose names differ only in alphabetic case.

Internally names of packages are strings, but the functions that require a package-name argument from the user accept either a symbol or a string. If you supply a symbol, its print-name is used, which has already undergone case conversion by the usual rules. If you supply a string, you must be careful to capitalize the string in the same way that the package's name is capitalized.

Note that |**Foo**|:|**Bar**| refers to a symbol whose name is "**Bar**" in a package whose name is "**Foo**". By contrast, |**Foo:Bar**| refers to a 7-character symbol with a colon in its name, and is interned in the current package. Following the convention used in the documentation for symbols, we show package names as being in lowercase, even though the name string is really in uppercase.

Invisible Packages

In addition to normal packages, there can be *invisible* packages. An invisible package has a name, but it is not entered into the system's table that maps package names to packages. An invisible package cannot be referenced via a qualified name (unless you set up a relative name for it) and cannot be used in such contexts as the **:use** keyword to **defpackage** and **make-package** (unless you pass the package object itself, rather than its name). Invisible packages are useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

Relative Package Names

See the section "Introduction to Package Names". In addition to the absolute package names (and nicknames) described there, packages can have *relative* names. If **p** is a relative name for package B, relative to package A, then in contexts where relative names are allowed and A is the contextually relevant package the name **p** can be used instead of **b**. The relative name mapping *belongs to* package A and defines a new name (**p**) *for* package B. It is important not to confuse the package that the name is relative to with the package that is named.

Relative names are established with the **:relative-names** and **:relative-names-for-me** options to **defpackage** and **make-package**. For example, to be able to refer to symbols in the **common-lisp** package with the prefix **lisp:** instead of **cl:** when they need a package prefix (for instance, when they are shadowed), you would use **:relative-names** like this:

```
(defpackage my-package (:use cl)
                       (:shadow error)
                       (:relative-names (lisp common-lisp)))


(let ((*package* (find-package 'my-package)))
  (print (list 'my-package::error 'cl:error)))
```

Then, when the current package (that is the value of ***package***) is **my-package**, you can refer to the **common-lisp** package as **lisp** both when reading and when printing. You can also use the **pkg-add-relative-name** function to establish a relative name. The **pkg-delete-relative-name** function removes a relative name.

There are two important differences between relative names and absolute names: relative names are recognized only in certain contexts, and relative names can "shadow" absolute names. One application for relative names is to replace one package by another. Thus if a program residing in package A normally refers to the **thermodynamics** package, but for testing purposes we would like it to use the

phlogiston package instead, we can give A a relative name mapping from the name **thermodynamics** to the **phlogiston** package. This relative name shadows the absolute name **thermodynamics**.

Another application for relative names is to ease the establishment of a family of mutually dependent packages. For example, if you have three packages named **algebra**, **rings**, and **polynomials**, these packages might refer to each other so frequently that you would like to use the nicknames **a**, **r**, and **p** rather than spelling out the full names each time. It would obviously be bad to use up these one-letter names in the system-wide space of package names; what if someone else has a program with two packages named **reasoning** and **truth-maintenance**, and would like to use the nicknames **r** and **t**? The solution to this name conflict is to make the abbreviated names be relative names defined in the **algebra**, **rings**, and **polynomials** packages. These abbreviations are seen by references emanating from those packages, and there is no conflict with other abbreviations defined by other packages.

An extension of the shadowing application for relative names is to set up a complete family of packages parallel to the normal one, such as **experimental-global** and **experimental-user**. Within this family of packages you establish relative name mappings so that the usual names such as **global** and **user** can be used. Certain system utility programs work this way.

When package A uses package B, in addition to inheriting package B's external symbols, any relative name mappings established by package B are inherited. In the event of a name conflict between relative names defined directly by A and inherited relative names, the inherited name is ignored. The results are unpredictable if two relative name mappings inherited from two different packages conflict.

The Lisp system does not itself use relative names, so a freshly booted Genera system contains no relative-name mappings.

Relative names are recognized in the following contexts:

- Qualified symbol names — The package name before the colon is relative to the package in which the symbol is being read (the value of the variable ***package***). The printer prefers a relative package name to an absolute package name when it prints a qualified symbol name.

- Package references in package-manipulating functions — For example, the package names in the **:use** option to **defpackage** and in the first argument to **use-package** can be relative names. All such relative names are relative to the value of the variable ***package***.

- Package arguments that default to the current package — The functions **intern**, **intern-local**, **intern-soft**, **intern-local-soft**, **unintern**, **export**, **unexport**, **import**, **shadow**, **shadowing-import**, **use-package**, and **unuse-package** all take an optional second argument that defaults (except in the case of **unintern**) to the current package. If supplied, this argument can be a package, an absolute name

of a package, or a relative name of a package. All such relative names are relative to the value of the variable ***package***.

Relative names are not recognized in "global" contexts, where there is no obvious contextual package to be relative to, such as:

- File attribute lists ("-*-" lines)
- Package names requested from you as part of error recovery, or in commands such as the Set Package (m-X) editor command.
- The **pkg-find-package** function (unless its optional third argument is specified).
- Package arguments to the **zl:mapatoms**, **zl:pkg-goto**, **describe-package**, and **pkg-kill** functions.
- Package specifiers in the **do-symbols**, **do-local-symbols**, and **do-external-symbols** special forms, and the **interned-symbols** and **local-interned-symbols loop** iteration paths.

When a package object is printed, if it has a relative name (relative to the value of ***package***) that differs from its absolute name, both names are printed.

Qualified Package Names

Introduction to Qualified Package Names

Often it is desirable to refer to an external symbol in some package other than the current one. You do this through the use of a *qualified name*, consisting of a package name, then a colon, then the name of the symbol. This causes the symbol's name to be looked up in the specified package, rather than in the current one. For example, **editor:buffer** refers to the external symbol named **buffer** of the package named **editor**, regardless of whether there is a symbol named **buffer** in the current package. If there is no package named **editor**, or if no symbol named **buffer** is present in **editor** or if **buffer** is an internal symbol of **editor**, an error is signalled.

On rare occasions, you might need to refer to an *internal* symbol of some package other than the current one. It is invalid to do this with the colon qualifier, since accessing an internal symbol of some other package is usually a mistake. See the section "Specifying Internal and External Symbols in Packages". However, this operation is valid if you use "**::**" as the separator in place of the usual colon. If the reader sees **editor::buffer**, the effect is exactly the same as reading **buffer** with ***package*** temporarily rebound to the package whose name is **editor**. This special-purpose qualifier should be used with caution.

Qualified names are implemented in the Lisp reader by treating the colon character (**:**) specially. When the reader sees one or two colons preceded by the name of

a package, it reads in the next Lisp object with ***package*** bound to that package. Note that the next Lisp object need not be a symbol; the printed representation of any Lisp object can follow a package prefix. If the object is a list, the effect is exactly as if every symbol in that list had been written as a qualified name, using the prefix that appears in front of the list. When a qualified name is among the elements of the list, the package name in the second package prefix is taken relative to the package selected by the first package prefix. The internal/external mode is controlled entirely by the innermost package prefix in effect.

Specifying Internal and External Symbols in Packages

To ease the transition for people whose programs are not yet organized according to the distinction between internal and external symbols, a package can be set up so that the "**:**" type of qualified name does the same thing as the "**::**" type. This is controlled by the package that appears before the colon, not by the package in which the whole expression is being read. To set this attribute of a package, use the **:colon-mode** keyword to **defpackage** and **make-package**. **:external** causes "**:**" to access only external symbols. See the section "Qualified Names of Symbols". See the section "Qualified Package Names as Interfaces". **:internal** causes "**:**" to behave the same as "**::**", accessing all symbols. Note that **:internal** mode is compatible with **:external** mode except in cases where an error would be signalled. The default mode is **:external** and all predefined system packages are created with this mode.

Qualified Package Names as Interfaces

See the section "How the Package System Allows Symbol Sharing". In the example of the blocks world and the robot arm, a program in the **blocks** package could call a function named **go-up** defined in the **arm** package by calling **arm:go-up**. **go-up** would be listed among the external symbols of **arm**, using **:export** in its **defpackage**, since it is part of the interface allowing the outside world to operate the arm. If the **blocks** program uses qualified names to refer to functions in the **arm** program, rather than sharing symbols as in the original example, then the possibility of name conflicts between the two programs is eliminated.

See the section "Example of the Need for Packages". Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function, it would simply call **arpa:allocate-pbuf**, assuming that function had been exported. If it was not exported (because **arpa** thought no one from the outside had any business calling it), the **chaos** program would call **arpa::allocate-pbuf**.

Qualified Names of Symbols

The printer uses qualified names when necessary. (The **princ** function, however, never prints qualified names for symbols.) The goal of the printer (for example, the **prin1** function) when printing a symbol is to produce a printed representation that the reader turns back into the same symbol. When a symbol that is accessible in the current package (the value of ***package***) is printed, a qualified name is not

used, regardless of whether the symbol is present in the package. This happens for one of three reasons: because this is its home package, is present because it was imported, or is not present but was inherited. When an inaccessible symbol is printed, a qualified name is used. The printer chooses whether to use "**:**" or "**::**" based on whether the symbol is internal or external and the **:colon-mode** of its home package. The qualified name used by the printer can be read back in and yields the same symbol. If the inaccessible symbol were printed without qualification, the reader would translate that printed representation into a different symbol, probably an internal symbol of the current package.

The qualified name used by the printer is based on the symbol's home package, not on the path by which it was originally read (which of course cannot be known). Suppose **foo** is an internal symbol of package A, has been imported into package B, and has then been exported from package B. If it is printed while ***package*** is neither A nor B, nor a package that uses B, the name printed is **a::foo**, not **b:foo**, because **foo**'s home package is A. This is an unlikely case, of course.

In addition to the simplest printed representation of a symbol, its name standing by itself, there are four forms of qualified name for a symbol. These are accepted by the reader and are printed by the printer when necessary; except when printing an uninterned symbol, the printer prints some printed representation that yields the same symbol when read. The following table shows the four forms of qualified name, assuming that the **foo** package specifies **:colon-mode :external**, as is the default. If **foo** specifies **:colon-mode :internal**, the first and second forms are equivalent.

foo:bar	When read, looks up **bar** among the external symbols of the package named **foo**. Printed when the symbol **bar** is external in its home package **foo** and is not accessible in the current package.
foo::bar	When read, interprets **bar** as if **foo** were the current package. Printed when the symbol **bar** is internal in its home package **foo** and is not accessible in the current package.
:bar	When read, interprets **bar** as an external symbol in the **keyword** package. Printed when the home package of the symbol **bar** is **keyword**.
#:bar	When read, creates a new uninterned symbol named **bar**. Printed when the symbol named **bar** has no home package.

Multilevel Qualified Package Names

Due to shadowing by relative names, a given package might sometimes be inaccessible. In this case a multilevel qualified name, containing more than one package prefix, can be used.

Suppose packages **moe**, **larry**, **curly**, and **shemp** exist. For its own reasons, the **moe** package uses **curly** as a relative name for the **shemp** package. Thus, when the current package is **larry** the printed representation **curly:hair** designates a symbol in the **curly** package, but when the current package is **moe** the same printed representation designates a symbol in the **shemp** package.

If the **moe** package is current and the symbol **hair** in the **curly** package needs to be read or printed, the printed representation **curly:hair** cannot be used since it refers to a different symbol. If **curly** had a nickname that is not also shadowed by a relative name it would be used, but suppose there is no nickname. In this case the only possible way to refer to that symbol is with a multilevel qualified name. **larry:curly:hair** would work, since the **larry:** escapes from the scope of **moe**'s relative name. The printer actually prefers to print **global:curly:hair** because of the way it searches for a usable qualified name.

Examples of Symbol Sharing Among Packages

See the section "How the Package System Allows Symbol Sharing". Consider again the example of the robot arm in the blocks world. Two separate programs, written by different people, interact with each other in a single Lisp environment. The arm-control program resides in a package named **arm**, and the blocks-world program resides in a package named **blocks**. The operation of the two programs requires them to interact. For example, to move a block from one place to another the **blocks** program calls functions in the **arm** program with names like **raise-arm**, **move-arm**, and **grasp**. To find the edges of the table, the **arm** program accesses variables of the **blocks** program.

Communication between the two programs requires that they both know about certain objects. Usually these objects are the sort that have names (for example, functions or variables). The names are symbols. Thus each program must be able to name some symbols and to know that the other program is naming the same symbols.

Let us consider the case of the function **grasp** in the arm-control program, which the blocks-world program must call in order to pick up a block with the arm. The **grasp** function is named by the symbol **grasp** in the **arm** package. Assume that we are not going to use either of the mechanisms (keywords and the **global** package) that make symbols available to *all* packages; we only want **grasp** to be shared between the two specific packages that need it. There are basically three ways provided by the package system for a symbol to be known by two separate programs in two separate packages.

1. If the **blocks** package *imports* the symbol **grasp** from the **arm** package, then both packages map the name **grasp** into the same symbol. The **blocks** package could be defined by:

```
(defpackage blocks
        (:import-from arm grasp))
```

2. The **arm** package can *export* the symbol **grasp**, along with whatever other symbols constitute its interface to the outside world. If the **blocks** package *uses* the **arm** package, then both packages again map the name **grasp** into the same symbol. The package definitions would look like:

```
(defpackage arm
        (:export grasp move-arm raise-arm ...))

(defpackage blocks
        (:use arm global))
```

   Note that the **blocks** package must explicitly mention that it is using the **global** package as well as the **arm** package, since it is not letting its **:use** clause default. The difference between this method (the export method) and the first method (the import method) is that the list of symbols that is to constitute the interface is associated with the **arm** package, that is, the package that *provides* the interface, not the package that *uses* the interface.

3. In the third method, we do not have the two packages map the same name into the same symbol. Instead we use a different, longer name for the symbol in the blocks program than the name used by the arm program. This makes it clear, when reading the text of the blocks program, which symbol references are connected with the interface between the two programs. These longer names are called *qualified names*. Again, the **arm** package defines the interface:

```
(defpackage arm
        (:export grasp move-arm raise-arm ...))
```

   A fragment of the blocks-world program might look like

```
(defun pick-up (block)
  (clear-top block)
  (arm:grasp (block-coordinates block :top))
  (arm:raise-arm))
```

   arm:grasp and **arm:raise-arm** are qualified names. **pick-up**, **block**, **clear-top**, and **block-coordinates** are internal symbols of the blocks-world program. **defun** is inherited from the **global** package. **:top** is a keyword. Note that although the two programs do not use the same names to refer to the same symbol, the names they use are related in an obvious way, avoiding confusion. The package system makes no provision for the same symbol to be named by two completely arbitrary names.

Consistency Rules for Packages

Package-related bugs can be very subtle and confusing: the program is not using the same symbols as you think it is using. The package system is designed with a number of safety features to prevent most of the common bugs that would otherwise occur in normal use. This might seem overprotective, but experience with earlier package systems has shown that such safety features are needed.

In dealing with the package system, it is useful to keep in mind the following consistency rules, which remain in force as long as the value of ***package*** is not changed by you or your code:

- *Read-Read consistency:* Reading the same print name always gets you the same (**eq**) symbol.

- *Print-Read consistency:* An interned symbol always prints as a sequence of characters that, when read back in, yields the same (**eq**) symbol.

- *Print-Print consistency:* If two interned symbols are not **eq**, then their printed representations cannot be the same sequence of characters.

These consistency rules remain true in spite of any amount of implicit interning caused by typing in Lisp forms, loading files, and so on. This has the important implication that results are reproducible regardless of the order of either loading files or typing in symbols. The rules can only be violated by explicit action: changing the value of ***package***, forcing some action by continuing from an error, or calling a function that makes explicit modifications to the package structure (**unintern**, for example).

To ensure that the consistency rules are obeyed, the system ensures that certain aspects of the package structure are chosen by conscious decision of the programmer, not by accidents such as which symbols happen to be typed in by a user. External symbols, the symbols that are shared between packages without being explicitly listed by the "accepting" package, must be explicitly listed by the "providing" package. No reference to a package can be made before it has been explicitly defined.

How Package Universes Work

Some dialects of Lisp have mutually incompatible constraints on package names. For example, both Zetalisp and Common-Lisp define a user package, but with incompatible meanings. Similarly, we have provided two different implementations of Common Lisp (in the CLtL and the Common-Lisp syntax), and both need a package named lisp.

Rather than make incompatible changes to our definitions of existing language dialects as we introduce new ones into Genera, we have created the notion of a package universe. This is the currently active global mapping from names of packages to package objects. So, for example, if you enter lisp:if in a Lisp Listener, you will get the symbol **cltl:if** if you are in CLtL syntax or **if** if you are in Common-Lisp syntax.

The active package universe is a direct function of the readtable. Each readtable has an associated universe. The binding of the *readtable* controls the interpretation of written syntax like lisp:if.

The implementation of a package universe is mediated by a "relative names" list for each syntax. Conceptually, this is like the relative names that packages have;

package relative names are searched first, then syntax relative names, then finally the global package namespace (if appropriate to the syntax; in some syntaxes, such as the CLtL-Only syntax, the global package namespace is never searched). At this time, there is no published interface for accessing or modifying the syntax relative names data structure, but it may help you to know that this is the mechanism.

Note, however, that some functions, such as the various find-package functions, are not sensitive to the dynamically current readtable. This is because they were read in a certain lexical environment but may be running in a hostile environment. For example, if you write

```
;-*- Syntax: Common-Lisp; ... -*-
(defun foo () (find-package "lisp"))
```

it should be possible to run the program foo sometime later in an environment where the readtable has been changed, without the program being broken. Thus, find-package or cltl:find-package or future-common-lisp:find-package will automatically use the proper package universe for their needs (Common-Lisp, CLtL, or ANSI-Common-Lisp, respectively).

Note that the following forms return the values shown, regardless of the binding of the readtable. These actions are not sensitive to the readtable since they are not read-related actions.

```
(CL:FIND-PACKAGE "LISP")
=> #<Package LISP 134103336> ; the Genera LISP package

(CLTL:FIND-PACKAGE "LISP")
=> #<Package COMMON-LISP-THE-LANGUAGE 14155367> ; the CLTL package

(CL:FIND-PACKAGE "COMMON-LISP")
=> #<Package LISP 134103336> ; the Genera LISP package

(FUTURE-COMMON-LISP:FIND-PACKAGE "COMMON-LISP")
=> #<Package FUTURE-COMMON-LISP 20031662500> ;the dpANS CL COMMON-LISP package
```

How Package Universes Work

Some dialects of Lisp have mutually incompatible constraints on package names. For example, both Zetalisp and Common-Lisp define a user package, but with incompatible meanings. Similarly, we have provided two different implementations of Common Lisp (in the CLtL and the Common-Lisp syntax), and both need a package named lisp.

Rather than make incompatible changes to our definitions of existing language dialects as we introduce new ones into Genera, we have created the notion of a package universe. This is the currently active global mapping from names of packages to package objects. So, for example, if you enter lisp:if in a Lisp Listener, you will get the symbol **cltl:if** if you are in CLtL syntax or **if** if you are in Common-Lisp syntax.

The active package universe is a direct function of the readtable. Each readtable has an associated universe. The binding of the *readtable* controls the interpretation of written syntax like lisp:if.

The implementation of a package universe is mediated by a "relative names" list for each syntax. Conceptually, this is like the relative names that packages have; package relative names are searched first, then syntax relative names, then finally the global package namespace (if appropriate to the syntax; in some syntaxes, such as the CLtL-Only syntax, the global package namespace is never searched). At this time, there is no published interface for accessing or modifying the syntax relative names data structure, but it may help you to know that this is the mechanism.

Note, however, that some functions, such as the various find-package functions, are not sensitive to the dynamically current readtable. This is because they were read in a certain lexical environment but may be running in a hostile environment. For example, if you write

```
;-*- Syntax: Common-Lisp; ... -*-
(defun foo () (find-package "lisp"))
```

it should be possible to run the program foo sometime later in an environment where the readtable has been changed, without the program being broken. Thus, find-package or cltl:find-package or future-common-lisp:find-package will automatically use the proper package universe for their needs (Common-Lisp, CLtL, or ANSI-Common-Lisp, respectively).

Note that the following forms return the values shown, regardless of the binding of the readtable. These actions are not sensitive to the readtable since they are not read-related actions.

```
(CL:FIND-PACKAGE "LISP")
=> #<Package LISP 134103336> ; the Genera LISP package

(CLTL:FIND-PACKAGE "LISP")
=> #<Package COMMON-LISP-THE-LANGUAGE 14155367> ; the CLTL package

(CL:FIND-PACKAGE "COMMON-LISP")
=> #<Package LISP 134103336> ; the Genera LISP package

(FUTURE-COMMON-LISP:FIND-PACKAGE "COMMON-LISP")
=> #<Package FUTURE-COMMON-LISP 20031662500> ;the dpANS CL COMMON-LISP package
```

Package Name-Conflict Errors

Introduction to Package Name-Conflict Errors

A fundamental invariant of the package system is that within one package any particular name can refer to only one symbol. A *name conflict* is said to occur when more than one candidate symbol exists and it is not obvious which one to choose. If the system does not always choose the same way, the read-read consistency rule would be violated. For example, some programs or data might have been

read in under a certain mapping of the name to a symbol. If the mapping changes to a different symbol, then additional programs or data are read, the two programs do not access the same symbol even though they use the same name. Even if the system did always choose the same way, a name conflict is likely to result in a different mapping from names to symbols than you expected, causing programs to execute incorrectly. Therefore, any time a name conflict occurs, an error is signalled. You can continue from the error and tell the package system how to resolve the conflict.

Note that if the same symbol is accessible to a package through more than one path, for instance as an external of more than one package, or both through inheritance and through direct presence in the package, there is no name conflict. Name conflicts only occur between distinct symbols with the same name.

See the section "Shadowing Symbols". As discussed there, the creator of a package can tell the system in advance how to resolve a name conflict through the use of *shadowing*. Every package has a list of shadowing symbols. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. A name conflict involving a shadowing symbol is always resolved in favor of the shadowing symbol, without signalling an error (except for one exception involving **import**). The **:shadow** and **:shadowing-import** options to **defpackage** and **make-package** can be used to declare shadowing symbols. The functions **shadow** and **shadowing-import** can also be used.

Checking for Package Name-Conflict Errors

Name conflicts are detected when they become possible, that is, when the package structure is altered. There is no need to check for name conflicts during every name lookup. The functions **use-package**, **import**, and **export** check for name conflicts.

Using a package makes the external symbols of the package being used accessible to the using package; each of these symbols is checked for name conflicts with the symbols already accessible.

Importing a symbol adds it to the internals of a package, checking for a name conflict with an existing symbol either present in the package or accessible to it. **import** signals an error even if there is a name conflict with a shadowing symbol, because two explicit directives from you are inconsistent.

Exporting a symbol makes it accessible to all the packages that use the package from which the symbol is exported. All of these packages are checked for name conflicts. **(export** *s p***)** does **(intern-soft** *s q***)** for each package *q* in **(package-used-by-list** *p***)**. Note that in the usual case of exporting symbols only during the initial definition of a package, there are no users of the package yet and the name-conflict checking takes no time.

intern does not need to do any name-conflict checking, because it never creates a new symbol if there is already an accessible symbol with the name given.

Note that the function **intern-local** can create a new symbol with the same name as an already accessible symbol. Nevertheless, **intern-local** does not check for name conflicts. This function is considered to be a low-level primitive and indiscriminate use of it can cause undetected name conflicts. Use **import**, **shadow**, or **shadowing-import** for normal purposes.

shadow and **shadowing-import** never signal a name-conflict error, because by calling these functions the user has specified how any possible conflict is to be resolved. **shadow** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the specified name is accessible, and if so whether it is directly present in the package or inherited; in the latter case a new symbol is created to shadow it. **shadowing-import** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the same name is accessible; if so it is shadowed by the new symbol, which implies that it must be uninterned (with **unintern**) if it was directly present in the package.

unuse-package, **unexport**, and **unintern** (when the symbol being removed is not a shadowing symbol) do not need to do any name-conflict checking, because they only remove symbols from a package; they do not make any new symbols accessible.

unintern of a shadowing symbol can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol **x**, and B and C each contain external symbols named **x**, then uninterning **x** from A reveals a name conflict between **b:x** and **c:x** if those two symbols are distinct. In this case **unintern** signals an error.

Resolving Package Name-Conflict Errors

Aborting from a name-conflict error leaves the original symbol accessible. Package functions always signal name-conflict errors before making any change to the package structure. Note: when multiple changes are to be made, for example when exporting a list of symbols, it is valid for each change to be processed separately, so that aborting from a name conflict caused by the second symbol in the list does not unexport the first symbol in the list. However, aborting from a name-conflict error caused by exporting a single symbol does not leave that symbol accessible to some packages and inaccessible to others; exporting appears as an atomic operation.

Continuing from a name-conflict error offers you a chance to resolve the name conflict in favor of either of the candidates. This can involve shadowing or uninterning. Another possibility that is offered to you is to merge together the conflicting symbols' values, function definitions, and property lists in the same way as **globalize**. This is useful when the conflicting symbols are not being used as objects, but only as names for functions (or variables, or flavors, for example). You are also offered the choice of simply skipping the particular package operation that would have caused a name conflict.

A name conflict in **use-package** between a symbol directly present in the using package and an external symbol of the used package can be resolved in favor of the first symbol by making it a shadowing symbol, or in favor of the second sym-

bol by uninterning the first symbol from the using package. The latter resolution is dangerous if the symbol to be removed is an external symbol of the using package, since it ceases to be an external symbol.

A name conflict in **use-package** between two external symbols inherited by the using package from other packages can be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol.

A name conflict in **export** between the symbol being exported and a symbol already present in a package that would inherit the newly exported symbol can be resolved in favor of the exported symbol by uninterning the other one, or in favor of the already present symbol by making it a shadowing symbol.

A name conflict in **export** or **unintern** due to a package inheriting two distinct symbols with the same name from two other packages can be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol, just as with **use-package**.

A name conflict in **import** between the symbol being imported and a symbol inherited from some other package can be resolved in favor of the symbol being imported by making it a shadowing symbol, or in favor of the symbol already accessible by not doing the **import**. A name conflict in **import** with a symbol already present in the package can be resolved by uninterning that symbol, or by not doing the **import**.

Good user-interface style dictates that **use-package** and **export**, which can cause many name conflicts simultaneously, first check for all of the name conflicts before presenting any of them to you. You can then choose to resolve all of them wholesale, or to resolve each of them individually, requiring considerable interaction but permitting different conflicts to be resolved different ways.

External-only Packages and Locking

A package can be *locked*, which means that any attempt to add a new symbol to it signals an error. Continuing from the error adds the symbol.

When reading from an interactive stream, such as a window, the error for adding a new symbol to a locked package does not go into the Debugger. Instead it asks you to correct your input, using the input editor. You cannot add a new symbol to a locked package just by typing its name; you must explicitly call **intern**, **export**, or **globalize**.

A package can be declared *external-only*. This causes any symbol added to the package to be exported automatically. Since exporting of symbols should be a conscious decision, when you create an external-only package it is automatically locked. Any attempt to add a new symbol to an external-only package signals an error because it is locked. If adding the symbol would cause a name conflict in some package that uses the package to which the symbol is being added, the error message mentions that fact. Continuing from the error adds the symbol anyway. In the event of name conflicts, appropriate proceed types for resolving name conflicts are offered.

To set up an external-only package, it can be temporarily unlocked and then the desired set of symbols can be interned in it. Unlocking an external-only package disables name-conflict checking, since the system (perhaps erroneously) assumes you know what you are doing. The **global** package is external-only and locked. Its contents are initialized when the system is built by reading files containing the desired symbols with ***package*** bound to the **global** package object, which is temporarily unlocked. The **system** package is external-only, locked, and initialized the same way.

Package Functions, Special Forms, and Variables

Packages are represented as Lisp objects. A package is a structure that contains various fields and a hash table that maps from names to symbols. Most of the structure field accessor functions for packages are only used internally by the package system and are not documented.

The **packagep** function returns **t** if its argument is a package object. This is equivalent to **(typep** *obj* **'package)**.

Many of the functions that operate on packages accept either an actual package or the name of a package. A package name can be either a string or a symbol.

Common Lisp naming convention uses a prefix of "**package-**" on names that do not already contain the word **package**. Many of the Zetalisp functions and variables associated with packages have names that begin with "**pkg-**". This naming convention is considered obsolete.

The Current Package

The current package is the value of the variable ***package***.

The following two functions change the current package:

zl:pkg-goto *pkg*	Make *pkg* the current package.
zl:pkg-bind *pkg body*	Evaluate *body* with *pkg* the current package.

Defining a Package

The **defpackage** special form is the preferred way to create a package. A **defpackage** form is treated as a *definition* form by the editor; hence the Edit Definition (`m-.`) command can find package definitions.

Typically you put a **defpackage** form in its own file, separate from the rest of a program's source code. The reason to use a separate file is that a package must be defined before it can be used. In order to compile, load, or edit your program, the package in which its symbols are to be read must already be defined. Typically the file containing the **defpackage** is read in the **user** package, while all the rest of the files of your program are read in your own private package.

When a large program consisting of multiple source files is maintained with the system system, one source file typically contains nothing but a **defpackage** form and a **defsystem** form. (Occasionally a few other housekeeping forms are present.) This file is called the *system declaration file*. The packages and systems built into the initial Lisp system are defined in two files: SYS:SYS;PKGDCL defines all the packages while SYS:SYS;PKGDCL defines all the systems. See the section "System Construction Tool".

In the simplest cases, where no nontrivial **defpackage** options are required, the **defpackage** form can be omitted and no separate file is required. All the information required to create your package is contained in the file attribute list of the file containing your program. See the section "Specifying Packages in Programs".

The **make-package** function is available as the primitive way to create package objects.

Functions for Defining a Package

defpackage *name options*	Define a package named *name*.
make-package *name*	Primitive subroutine called by **defpackage**. **defpackage** should be used in new programs.
pkg-kill *package*	Kill *package*.

Mapping Names to Symbols

The name of a symbol is a string, corresponding to the printed representation of that symbol with quoting characters removed. Mapping the name of a symbol into the symbol itself is called *interning*, for historical reasons. Interning is only meaningful with respect to a particular package, since packages are name-to-symbol mappings. Unless a package is explicitly specified, the current package is assumed.

There are four functions for interning: **intern**, **intern-soft**, **intern-local**, and **intern-local-soft**. Each function takes two arguments and returns two values. The arguments are a name and a package. The name can be a string or a symbol. The package argument can be a package, the name of a package as a string or a symbol, or **nil** or unsupplied, in which case the current package (the value of ***package***) is used by default.

The **-soft** functions do not create new symbols, but only find existing symbols. The other two functions add a new symbol to the package if no existing symbol with the specified name is found. When adding a new symbol, if the name argument is a string, a new symbol is created and its home package is made to be the specified package. If the name argument is a symbol, that symbol is used as the new symbol. If it has a home package, it is not changed, but if it does not have a home package its home package is set to the package to which it was just added.

The **-local** functions only look for symbols present in the package; they do not search through inherited symbols. The other two functions see all accessible symbols.

The first value is the symbol that was found or created, or **nil** if no symbol was found and a **-soft** function was called. The second value is a flag that takes on one of the following values:

nil	No preexisting symbol was found. If the function called was not a **-soft** version, a new internal symbol was added to the package.
:internal	An existing internal symbol was found to be present in the package.
:external	An existing external symbol was found to be present in the package.
:inherited	An existing symbol was found to be inherited by the package. This symbol is necessarily external in the package from which it was inherited, and cannot be external in the package being searched.

Note that the first value should not be used as a flag to detect whether or not a symbol was found, since the *false* value, **nil**, is a symbol. The second value must be used for this purpose. The **-soft** functions return both values **nil** if they do not find a symbol.

Note: interning is sensitive to case; that is, it considers two character strings different even if the only difference is one of uppercase versus lowercase (unlike most string comparisons elsewhere in Genera). Symbols are converted to uppercase when you type them in because the reader converts the case of characters in the printed representation of symbols; the characters are converted to uppercase before **intern** is ever called. So if you call **intern** with a lowercase **"foo"** and then with an uppercase **"FOO"**, you do not get the same symbol.

Functions That Map Names to Symbols

intern *string* &optional *(pkg* ***package****)*
: Finds or creates a symbol named *string* in *pkg*. Inherited symbols in *pkg* are included in the search for a symbol named *string*.

intern-local *sym* &optional *pkg*
: Finds or creates a symbol named *string* directly present in *pkg*. Symbols inherited by *pkg* from packages it uses are not considered, thus **intern-local** can cause a name conflict.

intern-soft *sym* &optional *pkg*
: Finds a symbol named *string* accessible to *pkg*, either directly present in *pkg* or inherited from a package it uses.

intern-local-soft *sym* &optional *pkg*
: Find a symbol named *string* directly present in *pkg*. Symbols inherited by *pkg* from packages it uses are not considered.

find-symbol *string* &optional *(pkg* ***package****)*
Searches *pkg* for the symbol *string*. It behaves like **intern** except that it never creates a new symbol.

find-all-symbols *str* Searches all packages for symbols named *string* and returns a list of them.

unintern *symbol* &optional *pkg*
Removes *sym* from *pkg* and from *pkg*'s shadowing-symbols list.

zl:intern *sym* &optional *pkg*
Finds or creates a symbol named *sym* accessible to the package *pkg*, either directly present in *pkg* or inherited from a package it uses.

zl:remob *sym* &optional *(pkg* **(symbol-package si:sym)***)*
In your new programs, we recommend that you use the function **unintern** which is the Common Lisp equivalent of the function **zl:remob**.

zl:remob *symbol* Removes *symbol*.

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.

Mapping Between Names and Packages

find-package *package* Returns the package object whose name is *package*.

package-name *package* Returns the name of the package object *package* as a string.

rename-package *pkg new-name*
Replaces the old name of *pkg* with *new-name*.

pkg-find-package *thing* Tries to interpret *thing* as a package.

zl:pkg-name *package* Gets the primary name of *package* as a string.

Package Iteration

The following functions allow you to operate on the symbols in a package.

do-symbols *variable body* Evaluates *body* with *variable* bound to each symbol accessible in current package in turn.

do-local-symbols *variable body* — Evaluates *body* with *variable* bound to those symbols present in the current package in turn.

do-external-symbols *variable body* — Evaluates *body* with *variable* bound to those symbols exported from the current package in turn.

do-all-symbols *variable body* — Evaluate *body* with *variable* bound to each symbol in all packages in turn.

zl:mapatoms *function* — Applies *function* to all symbols in current package.

zl:mapatoms-all *functions* — Applies *function* to all symbols in all packages.

See the section "Iteration Paths for **loop**". This section contains a discussion of the **interned-symbols** and **local-interned-symbols loop** iteration paths.

Interpackage Relations

pkg-add-relative-name *from-package name to-package*
Allows *from-package* to use *name* to refer to *to-package*.

pkg-delete-relative-name *from-package name*
Removes *name*.

package-use-list *package* — Lists other packages used by *package*.

package-used-by-list *package* — Lists the packages that use *package*.

use-package *packages-to-use* — Lets *packages-to-use* be used by the current package.

unuse-package *packages-to-unuse* — Remove *packages-to-unuse* from the use-list.

Functions that Import, Export, and Shadow Symbols

export *symbols* — Makes *symbols* external.

unexport *symbols* — Makes *symbols* internal.

import *symbols* — Makes *symbols* internal.

shadowing-import *symbols* — Makes *symbols* internal.

shadow *symbols* — Makes *symbols* internal.

package-external-symbols *package* — Lists external symbols.

package-shadowing-symbols *package*
Lists the shadowing symbols in *package*.

Package "Commands"

describe-package *package* — Displays the attributes of *package*.

where-is *pname* — Displays a description of each symbol named *pname* with its home package.

globalize *name* — Exports *name*.

There is a subtle pitfall in the interaction between **globalize** and the binary files output by the compiler. Because of this it is best to use a string, rather than a symbol, as the argument to **globalize** in files that are to be compiled. Suppose a file contains the following form at top level:

```
(eval-when (compile load eval)
  (globalize 'si:rumpelstiltskin))
```

If the file is loaded without being compiled, the form is read and evaluated in the obvious fashion. **rumpelstiltskin** is read as the symbol by that name in the **si** package, that symbol is passed to the **globalize** function, and the symbol is moved to the **global** package. Now suppose the file is compiled. Again **rumpelstiltskin** is read as the symbol by that name in the **si** package. The **eval-when** causes the compiler first to evaluate the **globalize** form, and then to place a representation of the form into its output file. But at the time the output file is being generated, the symbol **rumpelstiltskin** is global; the compiler no longer has any way to know that it came from the **si** package. When the binary file is loaded, it globalizes the symbol **rumpelstiltskin** in the current package, not the one in the **si** package as the programmer intended. Furthermore, if at compile time there was a **rumpelstiltskin** symbol in the current package, the compile-time **globalize** turns that symbol into a shadowing symbol. When the binary file is loaded, it tries to refer to the symbol **rumpelstiltskin** in the **global** package, which gets an error since the **global** package is locked. The same pitfall can arise without the use of **eval-when** if the file being compiled was previously loaded into the Lisp that compiled it, perhaps for test purposes.

System Packages

The following are some of the packages initially present in the Lisp world. New packages will be added to this list from time to time. The list is presented in "logical" order, with the most important or interesting packages first. A number of packages that are not of general interest have been omitted from the list for the sake of brevity.

cl, common-lisp or **common-lisp-global** — Contains the global symbols of Common Lisp. Inside of Common Lisp this package is called **lisp**. **common-lisp-global** does not inherit symbols from any other package.

zl or **global** — Contains the global symbols of the Zetalisp language, including function names, variable names, special form names, and so on. All symbols in **global** are supposed to be documented. **global** does not inherit symbols from any other package.

scl or **symbolics-common-lisp** — Contains the symbols that make up Symbolics extensions to

Common Lisp. **symbolics-common-lisp** uses **common-lisp-global**.

keyword Contains keyword symbols. **keyword** has a blank nickname so that keywords print as **:foo** rather than **keyword:foo**. **keyword** does not inherit symbols from any other package.

cl-user or **common-lisp-user**
The default package for user programs that do not have their own package. When first booted the Symbolics Lisp Machine uses the **cl-user** package to read expressions typed in by the user. Inside of Common Lisp, **cl-user** is called **user**. **cl-user** uses **common-lisp-global**.

zl-user or **zetalisp-user**
The default package for Zetalisp user programs. (See the section "Set Lisp Context Command".) Inside Zetalisp, **zl-user** is called **user**. **zl-user** uses **global**.

sys or **system** Contains symbols shared among various system programs. **system** is for symbols global to the Genera "operating system", while **common-lisp-global** and **global** are for symbols global to the Symbolics Common Lisp language.

si or **system-internals**
Most of the programs that implement the Lisp language and operating system are in the **system-internals** package. **system-internals** is one of the packages that uses **system**. The externally advertised symbols of these programs are in **system** or **global**.

compiler Contains the Lisp compiler. **compiler** is one of the packages that use **system**.

dbg or **debugger** Contains the condition system and the debugger. **debugger** is one of the packages that use **system**.

cp or **command-processor**
Contains the Command Processor.

flavor Contains some symbols related to Flavors

clos Exports symbols related to CLOS.

future-common-lisp
Contains symbols in the evolving ANSI Common Lisp standard including the symbols in the CLOS package that are part of the standard.

future-common-lisp-user
The ANSI Common Lisp equivalent of the **user** package, except that **future-common-lisp-user** uses only the **future-common-lisp** package; it does not include any Symbolics extensions to the language.

dw or **dynamic-windows**
Contains the dynamic window system. **dw** uses **system**.

tv Contains the window system window system substrate. **tv** is one of the packages that use **system**.

zwei Contains the editor and Zmail.

mailer Contains the Symbolics Store-and-Forward mailer.

fs or **file-system** Contains pathnames and the generic file access system. **file-system** is one of the packages that use **system**.

lmfs Contains the Genera file storage system. **lmfs** is one of the packages that use **system**.

format Contains the function **format** and its associated subfunctions.

hardcopy Contains the functions for sending output to printers.

net or **network** Contains the external interfaces to the generic network system. **network** is one of the packages that uses **system**. Each network implementation and network-related program has its own package, which uses **network**.

neti or **network-internals**
Contains the programs that implement the generic network system. **network-internals** uses **network** and **system**.

chaos Contains the Chaosnet control program. **chaos** is one of the packages that use **network** and **system**.

fonts Contains the names of all fonts. **fonts** does not inherit symbols from any other package.

The following variables have the most important packages as their values.

zl:pkg-global-package The **global** package.
sys:pkg-keyword-package The **keyword** package.
zl:pkg-system-package The **system** package.

Package-related Conditions

These are the most basic package-related conditions. There are other conditions built on these, but most programmers should not need to deal with them.

sys:package-error Basic error condition for packages.
sys:package-not-found Package lookup did not find a package with the specified name.
sys:external-symbol-not-found
The specified symbol is not external.
sys:package-locked There was an attempt to intern a symbol in a locked package.

sys:name-conflict Any sort of name conflict.

Multipackage Programs

Usually, each independent program occupies one package. But large programs, such as MACSYMA, are usually made up of a number of subprograms, and each subprogram might be maintained by a different person or group of people. We would like each subprogram to have its own namespace, since the program as a whole has too many names for anyone to remember. The package system can provide the same benefits to programs that are part of the same superprogram as it does to programs that are completely independent.

Putting each subprogram into its own package is easy enough, but it is likely that a fair number of functions and symbols should be shared by all of MACSYMA's subprograms. These would be internal interfaces between the different subprograms.

A package named **macsyma** can be defined and each of the internal interface symbols can be exported from it. Each subprogram of MACSYMA has its own package, which uses the **macsyma** package in addition to any other packages it uses. Thus the interface symbols are accessible to all subprograms, through package inheritance. These interface symbols typically get their function definitions, variable values, and other properties from various subprograms read into the various internal MACSYMA packages, although there is nothing wrong with also putting a subprogram directly into the **macsyma** package. This is similar to the way the Lisp system works; the **global** package exports a large number of symbols, which get their values, definitions, and so on from programs residing in other packages that use **global**, such as **system-internals** or **compiler**.

It is also often convenient for the **macsyma** package to supply relative names that can be used by the various subprograms to refer to each other's packages. This allows package name abbreviations to be used internally to MACSYMA without contaminating the external environment.

The system declaration file for MACSYMA would then look something like the following:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified names

;The integration package based on the Risch algorithm
(defpackage risch
            (:use macsyma global))
```

```
;The integration package based on pattern matching
(defpackage sin
            (:use macsyma global))

;Interface to the operating system.  This uses the SYSTEM package
;because it needs many system-dependent functions and constants.
;This package also has a local nickname because its primary name
;is so long.
(defpackage macsyma-system-interface
            (:relative-names-for-me (macsyma sysi))
            (:use macsyma system global))
```

You can break the interface symbols down into separate categories. For instance, you might want to separate internal symbols used only inside MACSYMA from symbols that are also useful to the outside world. The latter symbols clearly should be externals of the **macsyma** package. You could create an additional package named **macsyma-internals** that exports all the symbols that are interfaces between different subprograms of MACSYMA but are not for use by the outside world. In this case we would have:

```
(defpackage risch
            (:use macsyma-internals macsyma global))
```

A program in the outside world that needed to use parts of MACSYMA would either use qualified names such as **macsyma:solve** or would include **macsyma** in the **:use** option in its package definition.

The interface symbols can be broken down into even more categories. Each subpackage can have its own list of exported symbols, and can use whichever other subpackages it depends on. The subset of these exported symbols that are also useful to the outside world can be exported from the **macsyma** package as well. In this case our example system declaration file would look something like:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export solve integrate ...)
            (:colon-mode :external))    ;error-checking in qualified names

;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))
```

```
;The macsyma interpreter
(defpackage meval
            (:export meval mprint ...))
```

The symbol **integrate** exported by the **macsyma** package and the symbol **integrate** exported by the **risch** package are the same symbol, because **risch** inherits it from **macsyma**.

Sometimes you can get involved in forward references when setting up this sort of package structure. In the above example, **risch** needs to use **rat**, hence **rat** was defined first. If **rat** also needed to use **risch**, there would be no way to write the package definitions using only **defpackage**. In this case you can explicitly call **use-package** after both packages have been defined. For example:

```
;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))      ;also uses risch

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))

;Now complete the forward references
(use-package 'risch 'rat)
```

An analogous issue arises when using **:import**.

Now, the **risch** program and the **sin** program both do integration, and so it would be natural for each to have a function called **integrate**. From inside **sin**, **sin**'s **integrate** would be referred to as **integrate** (no prefix needed), while **risch**'s would be referred to as **risch::integrate** or as **risch:integrate** if **risch** exported it (which is likely). Similarly, from inside **risch**, **risch**'s own **integrate** would be called **integrate**, whereas **sin**'s would be referred to as **sin::integrate** or **sin:integrate**.

If **sin**'s **integrate** were a recursive function, you would refer to it from within **sin** itself, and would not have to type **sin:integrate** every time; you would just say **integrate**.

If the names **sin** and **risch** are considered to be too short to use up in the general space of package names, they can be made local abbreviations within MACSYMA's family of package through local names. The package definitions would be

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified names
```

```
;The integration package based on the Risch algorithm
(defpackage macsyma-risch-integration
            (:relative-names-for-me (macsyma risch))
            (:use macsyma global))

;The integration package based on pattern matching
(defpackage macsyma-pattern-integration
            (:relative-names-for-me (macsyma sin))
            (:use macsyma global))
```

From inside the **macsyma** package or any package that uses it the two integration functions would be referred to as **sin:integrate** and as **risch:integrate**. From anywhere else in the hierarchy, they could be called **macsyma:sin:integrate** and **macsyma:risch:integrate**, or **macsyma-pattern-integration:integrate** and **macsyma-risch-integration:integrate**.

Package Functions in the Common Lisp Package with Symbolics Common Lisp Extensions

Here is the function in the "Packages" chapter that has Symbolics Common Lisp extensions:

Function	*Extension(s)*
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*

Input/Output Facilities

How the Reader Works

The purpose of the reader is to accept characters, interpret them as the printed representation of a Lisp object, and return a corresponding Lisp object. The reader cannot accept everything that the printer produces; for example, the printed representations of arrays (other than strings), compiled code objects, closures, stack groups, and so on cannot be read in. However, it has many features that are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently used unwieldy constructs.

In general, the reader operates by recognizing *tokens* in the input stream. Tokens can be self-delimiting or can be separated by delimiters such as whitespace. A token is the printed representation of an atomic object such as a symbol or a number, or a special character such as a parenthesis. The reader reads one or more tokens until the complete printed representation of an object has been seen, and then constructs and returns that object.

What the Reader Recognizes

The chapters on individual data types describe how the reader recognizes objects of those types. See the section "Data Types".

The relevant sections within that chapter are:

"How the Reader Recognizes Characters"
"How the Reader Recognizes Complex Numbers"
"How the Reader Recognizes Floating-Point Numbers"
"How the Reader Recognizes Integers"
"How the Reader Recognizes Lists"
"How the Reader Recognizes Numbers"
"How the Reader Recognizes Rational Numbers"
"How the Reader Recognizes Ratios"
"How the Reader Recognizes Sequences"
"How the Reader Recognizes Strings"
"How the Reader Recognizes Symbols"

How the Reader Recognizes Macro Characters

Certain characters are defined to be macro characters. When the reader sees one of these, it calls a function associated with the character. This function reads whatever syntax it likes and returns the object represented by that syntax. Macro characters are always token delimiters; however, they are not recognized when quoted by slash or vertical bar, nor when inside a string. Macro characters are a syntax-extension mechanism available to the user. Lisp comes with several predefined macro characters:

Quote (')	An abbreviation to make it easier to put constants in programs. *'foo* reads the same as **(quote** *foo***)**.
Semicolon (;)	Used to enter comments. The semicolon and everything up through the next carriage return are ignored. Thus a comment can be put at the end of any line without affecting the reader.
Backquote (`)	Makes it easier to write programs to construct lists and trees by using a template. See the section "Backquote-Comma Syntax".
Comma (,)	Part of the syntax of backquote. It is invalid if used other than inside the body of a backquote. See the section "Backquote-Comma Syntax".
Sharp sign (#)	Introduces a number of other syntax extensions. See the section "Sharp-sign Reader Macros". Unlike the preceding characters, sharp sign is not a delimiter. A sharp sign in the middle of a symbol is an ordinary character.

The function **zl:set-syntax-macro-char** can be used to define your own macro characters.

Reader macros that call a read function should call **si:read-recursive**.

si:read-recursive *stream*

si:read-recursive should be called by reader macros that need to call a function to read. It is important to call this function instead of **zl:read** in macros that are written in Zetalisp but used by the Common Lisp readtable. In particular, this function must be called by macros used in conjunction with the Common Lisp #*n*= and #*n*# syntaxes.

Sharp-sign Reader Macros

The reader's syntax includes several abbreviations introduced by sharp sign (#) take the general form of a sharp sign, a second character that identifies the syntax, and following arguments. Certain abbreviations allow a decimal number or certain special "modifier" characters between the sharp sign and the second character.

The function **zl:set-syntax-#-macro-char** can be used to define your own sharp-sign abbreviations.

#\ or #/

#*x* (or #/*x* in Zetalisp) reads in as the character *x*. For example, **#\a**. This is the recommended way to include character constants in your code. Note that the backslash causes this construct to be parsed correctly by the editor.

As in strings, upper- and lowercase letters are distinguished after #\. Any character works after #\, even those that are normally special to **read**, such as parentheses.

#*name* (or #/*name*) reads in as the name for the nonprinting character symbolized by *name*. A large number of character names are recognized. See the section "Special Character Names". For example, **#\return** reads in as a character, being the character code for the Return character in the Genera character set. In general, the names that are written on the keyboard keys are accepted. The abbreviations **#\cr** for **#\return** and **#\sp** for **#\space** are accepted and generally preferred, since these characters are used so frequently. The page separator character is called **#\page**, although **#\form** and **#\clear-screen** are also accepted since the keyboard has one of those legends on the page key. The rules for reading *name* are the same as those for symbols; thus upper- and lowercase letters are not distinguished, and the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

When the system types out the name of a special character, it uses the same table as the #\ reader; therefore, any character name typed out is acceptable as input.

#\ (or #/) can also be used to read in the names of characters that have control and meta bits set. The syntax looks like **#\control-meta-B** to get a "B" character with the control and meta bits set. You can use any of the prefix bit names **shift**, **control**, **meta**, **hyper**, and **super**. They can be in any order, and upper- and lowercase letters are not distinguished. The last hyphen can be followed by a single character, or by any of the special character names normally recognized by #\. If it is a single character, it is treated the same way the reader normally treats characters in symbols; if you want to use a lowercase character or a special character such as a parenthesis, you must precede it with a backslash character. Examples: **#\hyper-super-A**, **#\meta-hyper-roman-i**, **#\ctrl-meta-\(**.

Note that the specification of Common Lisp in Steele's *Common Lisp: the Language* states that the printed representation of the character **#\a** with control and meta bits on would be **#\control-meta-\a**. In the Symbolics implementation, there is no character corresponding to a lowercase a with the control and meta bits on. Therefore, the printed representation of **#\control-meta-\a** is **#\control-meta-\A**.

#^

#^*x*
Generates Control-*x*. In Maclisp *x* is converted to uppercase and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by **tyi** if the user holds down the control key and types *x*.

NOTE: #^ Reader Macro is supported in Zetalisp only.

#'

#'*foo* is an abbreviation for **(function** *foo***)**. *foo* is the printed representation of any object. This abbreviation can be remembered by analogy with the ' macro character, since the **function** and **quote** special forms are somewhat analogous.

#,

#,*foo* evaluates *foo* (the printed representation of a Lisp form) at read time, unless the compiler is doing the reading, in which case it is arranged that *foo* be evaluated when the FASL file is loaded. This is a way, for example, to include in your code complex list-structure constants that cannot be written with **quote**. Note that the reader does not put **quote** around the result of the evaluation. You must do this yourself, typically by using the ' macro-character. An example of a case where you do not want **quote** around it is when this object is an element of a constant list.

#.

#.*foo* evaluates *foo* (the printed representation of a Lisp form) at read time, regardless of who is doing the reading.

#:

#:*name* reads *name* as an uninterned symbol. It always creates a new symbol. Like all package prefixes, **#:** can be followed by any expression. Example: **#:(a b c)**.

#b

#b*rational* reads *rational* (an integer or a ratio) in binary (radix 2). Examples:

```
#B1101 <=> 13.
#B1100\100 <=> 3
```

#o

#o *number* reads *number* in octal regardless of the setting of **ibase**. Actually, any expression can be prefixed by **#o**; it is read with **ibase** bound to 8.

#x

#x *number* reads *number* in radix 16. (hexadecimal) regardless of the setting of *ibase*. As with **#o**, any expression can be prefixed by **#x**. The *number* can contain embedded hexadecimal "digits" A through F as well as the 0 through 9. See the section "Reading Integers in Bases Greater Than 10".

#r

#*radix***R** *number* reads *number* in radix *radix* regardless of the setting of **ibase**. As with **#o**, any expression can be prefixed by **#***radix***R**; it is read with **ibase** bound to *radix*. *radix* must consist of only digits, and it is read in decimal. *number* can consist of both numeric and alphabetic digits, depending upon *radix*.

For example, **#3R102** is another way of writing **11.** and **#11R32** is another way of writing **35.**

#Q

#Q *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace).

Note: **#Q** is supported only for Zetalisp.

#M

#M *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace).

Note: **#M** is supported only for Zetalisp.

#N

#N *foo* reads as *foo* if the input is being read into NIL or compiled to run in NIL, otherwise it reads as nothing (whitespace). Also, during the reading of *foo*, the reader temporarily defines various NIL-compatible sharp-sign reader macros (such as **#!** and **#'**) in order to parse the form correctly, even though it is not going to be evaluated.

Note: **#N** is supported only for Zetalisp.

#+

This abbreviation provides a read-time conditionalization facility similar to, but more general than, that provided by **#M**, **#N**, and **#Q**. It is used as **#+***feature form*. If *feature* is a symbol, then this is read as *form* if **(status feature** *feature***)** is **t**. If **(status feature** *feature***)** is **nil**, then this is read as whitespace. Alternately, *feature* can be a boolean expression composed of **and**, **or**, and **not** operators and symbols representing items which can appear on the **(status features)** list. **(or lispm amber)** represents evaluation of the predicate **(or (status feature lispm) (status feature amber))** in the read-time environment. Note that the feature names are found in the keyword package.

For example, **#+lispm** *form* makes *form* exist if being read by Symbolics Common Lisp, and is thus equivalent to **#Q** *form*. Similarly, **#+maclisp** *form* is equivalent to **#M** *form*. **#+(or lispm nil)** *form* makes *form* exist on either Symbolics Common Lisp or in NIL. Note that items can be added to the **(status features)** list by means of **(sstatus feature** *feature***)**, thus allowing the user to selectively interpret or compile pieces of code by parameterizing this list. See the special form **zl:status**. See the special form **zl:sstatus**.

#-

#-*feature form* is equivalent to **#+(not** *feature***)** *form*. Note that the feature names are found in the keyword package.

#|

#| begins a comment for the Lisp reader. The reader ignores everything until the next |#, which closes the comment. Note that if the |# is inside a comment that begins with a semicolon, it is *not* ignored; it closes the comment that began with the preceding #|. #| and |# can be on different lines, and #|**...**|# pairs can be nested.

Using #|**...**|# always works for the Lisp reader. The editor, however, does not understand the reader's interpretation of #|**...**|#. Instead, the editor retains its knowledge of Lisp expressions. Symbols can be named with vertical bars, so the editor (not the reader) behaves as if #|**...**|# is the name of a symbol surrounded by pound signs, instead of a comment.

Note: Use #||**...**||# instead of #|**...**|# to comment out Lisp code.

The reader views #||**...**||# as a comment: the comment prologue is #|, the comment body is |**...**|, and the comment epilogue is |#. The editor, however, interprets #||**...**||# as a pound sign (#), a symbol with a zero-length print name (||), Lisp code (**...**), another symbol with a zero length print name (||), and a stray pound sign (#). Therefore, inside a #||**...**||#, the editor commands that operate on Lisp code, such as balancing parentheses and indenting code, work correctly.

#<

This is not valid reader syntax. It is used in the printed representation of objects that cannot be read back in. Attempting to read a #< causes an error.

#◊

#◊ turns infix expression syntax into regular Lisp code. It is intended for people who like to use traditional arithmetic expressions in Lisp code. It is not intended to be extensible or to be a full programming language. We do not intend to extend it into one.

```
(defun my-add (a b)
  #◊a+b◊)
```

The quoting character is backslash. It is necessary for including special symbols (such as -) in variable names.

! reads one Lisp expression, which can use this reader-macro inside itself.

#◊ supports the following syntax:

Delimiters — Begin the reader macro with #◊, complete it with ◊.

```
#◊A+b-c◊
```

Escape characters — Special characters in symbol names must be preceded with backslash (\). You can escape to normal Lisp in an infix expression; precede the Lisp form with exclamation point (**!**).

Symbols — Start symbols with a letter. They can contain digits and underscore characters. Any other characters need to be quoted with \.

Operators — It accepts the following classes of operators. Arithmetic operator precedence is like that in FORTRAN and PL/I.

Operator	*Infix*	*Lisp Equivalent*
Assignment	x : y	(setf x y)
Functions	f(x,y)	(f x y) -- also works for defstruct accessors, and so on.
Array ref	a[i,j]	(aref a i j)
Unary ops	+ - not	*same*
Binary ops	+ - * / ^ = ≠	*same*
	< ≤ > ≥ and or	*same*
Conditional	if p then c	(if p c)
	if p then c else a	(if p c a)
Grouping:	(a, b, c)	(progn a b c) -- even works for (1+2)/3

The following example shows matrix multiplication using an infix expression.

```
(defun matrix-multiply (a b)
  (let ((n (array-dimension-n 2 a)))
    (unless (= n (array-dimension-n 1 b))
      (ferror "Matrices ~S and ~S do not have compatible ~
               dimensions") a b)
    (let ((d1 (array-dimension-n 1 a))
          (d2 (array-dimension-n 2 b)))
      (let ((c #◊ make\-array(list(d1, d2), !:type, art\-float)◊ ))
        (dotimes (i d1)
          (dotimes (j d2)
            #◊ c[i,j] : !(loop for k below n
                               sum #◊ a[i,k]*b[k,j] ◊)◊))
        c))))
```

The line containing the infix expression could also have been written like this:

```
(let ((sum 0))
  (dotimes (k n) #◊ sum:sum+a[i,k]*b[k,j] ◊)
  #◊ c[i,j]:sum ◊)
```

Special Character Names

The following are the recognized special character names in alphabetical. These names can be used after a #\ to get the character code for that character. Most of these characters type out as this name enclosed in a lozenge.

The special characters are:

```
Abort
Alpha
And-sign
Back-Space
Beta
Center-Dot
Circle
Circle-Plus
Circle-X
Clear-Input
Close
Close-parenthesis
Complete
Delta
Double-Arrow
Down-Arrow
Down-Horseshoe
End
Eow-Down
Epsilon
```

```
Equal-sign
Equivalence
Escape
Existential-Quantifier
Function
Gamma
Greater-Or-Equal
Greater-sign
Help
Infinity
Integral
Lambda
Left-Arrow
Left-Horseshoe
Less-Or-Equal
Less-sign
Line
Lozenge
Minus-sign
network
Not-Equals
Not-Sign
Null
Open
Open-parenthesis
Or-sign
Page
Partial-Delta
Pi
Plus-Minus
Plus-sign
Refresh
Resume
Return
Right-horseshoe
Rightarrow
Rubout
Scroll
Select
Space
Square
Suspend
Symbol-Help
Tab
Triangle
Universal-Quantifier
Up-Arrow
Up-Horseshoe
```

The following are special characters sometimes used to represent single and double mouse clicks. The buttons can be called either **l**, **m**, **r** or **1**, **2**, **3** depending on stylistic preference. The list below represents single mouse clicks (preceed with an **sh-** to represent double mouse clicks).

Mouse-L-1=Mouse-1-1	Mouse-L-2=Mouse-1-2
Mouse-M-1=Mouse-2-1	Mouse-M-2=Mouse-2-2
Mouse-R-1=Mouse-3-1	Mouse-R-2=Mouse-3-2

The Readtable

A data structure called the ***readtable*** (or readtable) is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard Lisp meanings to all the characters, but you can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the symbol ***readtable***.

You can program the reader by changing the readtable in any of three ways:

- The syntax of a character can be set to one of several predefined possibilities.
- A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function that is called when the character is read.
- You can create a completely new readtable, using the readtable compiler (sys:io;rtc) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.

Functions and Variables Related to Readtables

copy-readtable &optional *(from-readtable* ***readtable***) *to-readtable*
A copy is made of *from-readtable*, which defaults to the current readtable.

zl:copy-readtable &optional *(a-readtable* **zl:readtable**) *another-readtable*
from-readtable, which defaults to the current readtable, is copied. If *to-readtable* is unsupplied or **nil**, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copy.

readtablep *object* Returns **t** if *object* is a readtable, otherwise it returns **nil**.

readtable The current readtable.

zl:readtable ***readtable*** is the Common Lisp equivalent of **zl:readtable**.

si:initial-readtable The value of **si:initial-readtable** is the initial standard readtable.

Functions That Change Character Syntax

set-syntax-from-char *to-char from-char* &optional *(to-readtable* ***readtable****) from-readtable*
This makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*.

zl:set-syntax-from-char *char known-char* &optional *(a-readtable* **zl:readtable***) (known-readtable* **si:initial-readtable***)*
Makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*.

set-character-translation *char* &optional *value (a-readtable* **zl:readtable***)*
Changes *readtable* so that *from-char* is translated to *to-char* upon read-in, when *readtable* is the current readtable.

zl:set-syntax-from-description *char description* &optional *(a-readtable* **zl:readtable***)*

Sets the syntax of *char* in *readtable* to be that described by the symbol *description*.

Functions That Change Characters Into Macro Characters

make-dispatch-macro-character *char* &optional *non-terminating-p (a-readtable* ***readtable****)*
Causes *char* to be a dispatching macro character in *readtable*.

set-dispatch-macro-character *disp-char sub-char function* &optional *(a-readtable* ***readtable****)*
Causes *function* to be called when the *disp-char* followed by *sub-char* is read.

get-dispatch-macro-character *disp-char sub-char* &optional *(a-readtable* ***readtable****)*
Returns the macro-character function for *sub-char* under *disp-char*, or **nil** if there is no function associated with sub-char.

set-macro-character *char function* &optional *non-terminating-p (a-readtable* ***readtable****)*
Causes *char* to be a macro character that causes *function* to be called when it is seen by the reader.

get-macro-character *char* &optional *(a-readtable* ***readtable****)*
Returns two values: the function associated with *char*, and the value of the *non-terminating-p* flag.

zl:set-syntax-macro-char *char function* &optional *(a-readtable* **zl:readtable***) non-terminating*
Changes *readtable* so that *char* is a macro character.

zl:set-syntax-#-macro-char *char function* &optional *(a-readtable* **zl:readtable***)*
Causes *function* to be called when #*char* is read.

Readtable Functions for Maclisp Compatibility

zl:setsyntax *char magic more-magic*
The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*.

zl:setsyntax-sharp-macro *char type fun* &optional *(rdtbl* **zl:readtable***)*
This exists only for Maclisp compatibility; **zl:set-syntax-#-macro-char** is preferred.

Input Functions

Most of these functions take an optional argument to specify the stream from which to take characters, called *input-stream* in Common Lisp and *stream* in Zetalisp. *input-stream* is the stream from which the input is to be read; if unsupplied it defaults to the value of ***standard-input***. The special pseudostreams **nil** and **t** are also accepted. **nil** means the value of ***standard-input*** (that is, the default) and **t** means the value of ***terminal-io*** (that is, the interactive terminal). See the section "Introduction to Streams". These functions also take optional end-of-file arguments. In Common Lisp, the options are *eof-error-p* and *eof-value*. *eof-error-p* controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is **t** (the default), an error will be signalled at the end of a file. If it is **nil**, then no error is signalled, and instead the function returns *eof-value*.

In Zetalisp, if no *eof-option* argument is supplied, an error is signalled. If there is an *eof-option*, it is the value to be returned. Note that an *eof-option* of **nil** means to return **nil** if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

Functions such as **read** that read an "object" rather than a single character always signal an error, regardless of *eof-error-p* or *eof-option*, if the file ends in the middle of an object. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** complains. If a file ends in a symbol or a number immediately followed by end-of-file, **read** reads the symbol or number successfully and when called again, sees the end-of-file and obeys *eof-error-p*. Any whitespace at the end of a file is not considered to be an object. A file that ends in whitespace will only signal an error if *eof-error-p* is **t**.

The optional argument *recursive-p* specifies whether a call to an input function is a "top-level" or an embedded call. If the argument is **t**, the function call is treated recursively. The context of the *recursive-p* argument is most often found in the definition of a macro character. For example, if the single quote macro is defined as:

```
(set-macro-character
  #\'
  #'(lambda (stream char)
      (declare (ignore char))
      (list 'quote (read stream))))
```

then an expression such as

```
(cons '#1=(a b c) '(#1# . 1 2 3))
```

will not be read properly. The recursive call to **read** by the first ' is not treated recursively and the label is lost when the expression

```
'#1=(a b c)
```

is completed. The second ' is also interpreted as a "top-level" call to **read**. When the pointer **#1#** is reached and there is no associated label, the Lisp Reader traps the error and offers the opportunity to correct the input.

The single quote macro below is defined properly. Note the use of *recursive-p* in the call to **read**:

```
(set-macro-character
  #\'
  #'(lambda (stream char)
      (declare (ignore char))
      (list 'quote (read stream :recursive-p t))))
```

Now, when the expression

```
(cons '#1=(a b c) '(#1# . 1 2 3))
```

is evaluated, the recursive calls to **read** are properly recognized.

recursive-p also determines the kind of error that is signalled if *eof-error-p* is not **nil**. If *recursive-p* is not **nil** and the end-of-file (EOF) is encountered, the EOF is considered to have occured in the middle of an object, and an error is signaled. If *recursive-p* is **nil** and the EOF is encountered, the EOF is considered to have occured between objects, and no error is signalled.

Note that all of these functions except **zl:readline-no-echo** echo their input if used on an interactive stream (one that supports the **:input-editor** operation. The functions that input more than one character at a time (**zl:read**, **zl:readline**) allow the input to be edited using rubout. **zl:tyipeek** echoes all of the characters that were skipped over if **zl:tyi** would have echoed them; the character not removed from the stream is not echoed either.

Input Functions That Work on Streams

The following functions work on input or bidirectional streams:

read &optional *input-stream (eof-errorp* **t***) eof-value recursive-p*

Reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object.

zl:read &optional *(stream* **zl:standard-input***) eof-option*

Reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object.

sys:read-character &optional *stream* &rest *keywords* &key *(:fresh-line* **t***) :any-tyi :no-hang :eof (:notification* **t***) :prompt :help (:refresh* **t***) (:suspend* **t***) (:abort* **t***) :timeout :input-wait :input-block :input-wait-handler :whostate :status :presentation-context*
This function displays notifications and help messages and re-prompts at appropriate times.

zl:tyi &optional *stream eof-option*
Inputs one character from *stream* and returns it.

sys:read-for-top-level &optional *(stream* **zl:standard-input***) eof-option*
Differs from **zl:read** only in that it ignores close parentheses seen at top level, and it returns the symbol **si:eof** if the stream reaches end-of-file if you have not supplied an *eof-option* (instead of signalling an error as **zl:read** would).

zl:read-expression &optional *stream* &key *:completion-alist :completion-delimiters (:presentation-type* **'sys:expression***)*
Like **sys:read-for-top-level** except that if it encounters a top-level end-of-file, it just beeps and waits for more input.

zl:read-form &optional *stream* &key *(:edit-trivial-errors-p* **zl:*read-form-edit-trivial-errors-p***) (:completion-alist* **zl:*read-form-completion-alist***) (:completion-delimiters* **zl:*read-form-completion-delimiters***) (:environment* **si:*read-form-environment***) (:presentation-type* **'sys:form***)*
Like **zl:read-expression** except that it assumes that the returned value will be given immediately to **eval**.

read-or-end &optional *(stream* **zl:standard-input***) reader*
Like **zl:read-expression** except that if it is reading from an interactive stream and the user presses END as the first character or the first character after only whitespace characters, it returns two values, **nil** and **:end**.

zl:read-or-character &optional *delimiters stream reader* &rest *reader-args* &key *:presentation-type* &allow-other-keys
Like **zl:read-expression**, except that if it is reading from an interactive stream and the user types one of the *delimiters* as the first character or the first character after only whitespace characters, it returns four values: **nil**, **:character**, the character code of the delimiter, and any numeric argument to the delimiter. If it encounters any nonwhitespace characters, it calls the *reader* function with an argument of *stream* to read the input.

zl:read-and-eval &optional *stream (catch-errors* **t***)* &key *(:environment* **si:*read-form-environment***)*
Calls **zl:read-expression** to read a form, without completion.

read-line &optional *input-stream (eof-errorp* **t***) eof-value recursive-p*
Reads in a line of text.

zl:readline &optional *(stream* **zl:standard-input***) eof-option*
This function is usually used to get a line of input from the user.

read-line-trim &optional *input-stream (eof-errorp* **t***) eof-value recursive-p*
Reads in a line of text and returns it as the first value after trimming leading and trailing whitespace, that is, spaces and tabs.

zl:readline-trim &optional *(stream* **zl:standard-input***) (eof-option* **'si:no-eof-option***)*
like **zl:readline** except that **zl:readline-trim** trims leading and trailing whitespace — spaces and tabs — from string input.

zl:readline-or-nil &optional *(stream* **zl:standard-input***) (eof-option* **'si:no-eof-option***)*

Reads in a line of text.

read-line-no-echo &optional *stream* &rest *keywords* &key *(:terminators* **'(#\return #\line #\end)***) :full-rubout (:notification* **t***) :prompt :help*
Reads a line of input from *stream* without echoing the input, and returns the input as a string, without the terminating character.

zl:readline-no-echo &optional *stream* &rest *keywords* &key *(:terminators* **'(#\return #\line #\end)***) :full-rubout (:notification* **t***) :prompt :help*
Reads a line of input from *stream* without echoing the input, and returns the input as a string, without the terminating character.

read-delimited-list *char* &optional *stream recursive-p*
Reads objects from *stream* until the next character after an object's representation is *char*.

read-delimited-string *delimiters* &optional *stream eof-error-p eof-value* &rest *make-array-args*
The characters read up to the delimiter are returned as a string.

zl:read-delimited-string &optional *(delimiters* **#\end***) (stream* **zl:standard-input***) eof* &rest *(make-array-args (quote ((100 :type sys:art-string))))*
Characters are read from *stream* until one of the delimiter characters is encountered.

read-preserving-whitespace &optional *input-stream (eof-errorp* **t***) eof-value recursive-p*
Used for some specialized situations where it is desirable to determine precisely what character terminated the extended token.

read-char &optional *input-stream (eof-errorp* **t***) eof-value recursive-p*
Reads one character from *input-stream*, and returns it as a character object.

zl:readch &optional *stream eof-option*
Provided for Maclisp compatibility only. **zl:readch** is just like **zl:tyi**, except that instead of returning a character object, it returns a symbol whose print name is the character read in.

read-char-no-hang &optional *input-stream (eof-errorp* **t***) eof-value recursive-p*
Performs the same operation as **read-char**, but if it would be necessary to wait in order to get a character (as from a keyboard), **nil** is immediately returned without waiting.

unread-char *character* &optional *input-stream*
Puts *character* onto the front of *input-stream*.

read-byte *binary-input-stream* &optional *(eof-errorp* **t***) eof-value*
Reads one byte from *binary-input-stream* and returns it in the form of an integer.

peek-char &optional *peek-type input-stream (eof-errorp* **t***) eof-value recursive-p*
Returns the next character to be read from *input-stream*, without actually removing it from the stream. (This is the default behavior, which can be modified by the *peek-type* argument.)

zl:tyipeek &optional *peek-type stream eof-option*
Provided mainly for Maclisp compatibility; the **:tyipeek** stream operation is usually preferred.

clear-input &optional *input-stream*
Clears any buffered input associated with *input-stream*.

listen &optional *input-stream*
Returns **t** if there is a character immediately available from *input-stream*, and otherwise it returns **nil**.

Non-stream Input Functions

The following are input functions that do not operate on streams:

read-from-string *string* &optional *(eof-errorp* **t***) eof-value* &key *(:start* **0***) :end :preserve-whitespace*
Reads a Lisp object from a string

zl:read-from-string *string* &optional *(eof-option* **'si:no-eof-option***) (start* **0***) end (preserve-whitespace* **zl:read-preserve-delimiters***)*
The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned.

parse-integer *string* &key *(:start* **0***) :end (:radix* **10***) :junk-allowed (:sign-allowed* **t***)*
Examines the substring of *string* delimited by **:start** and **:end**, skips over whitespace, and then attempts to parse an integer.

zl:readlist *si:*iolst* Provided mainly for Maclisp compatibility. The elements in *char-list* are given successively to the reader, and the Lisp object built by the reader is returned.

See the function **with-input-from-string**.

Read Control Variables

There are a number of reader variables that affect the performance of read functions.

read-suppress When the value of ***read-suppress*** is **nil**, the Lisp reader operates normally.

zl:read-preserve-delimiters
Useful for certain reader macros or special syntaxes.

zl:*read-form-edit-trivial-errors-p*
If not **nil**, **zl:read-form** checks for two kinds of errors; if a symbol is read, it checks whether the symbol is bound.

zl:*read-form-completion-alist*
If not **nil**, **zl:read-form** sets up `COMPLETE` and `c-?` as input editor commands.

zl:*read-form-completion-delimiters*
If **zl:*read-form-completion-delimiters*** is **nil**, the entire text of the current symbol is a single "chunk".

Printed Representation

Printed Representation Of Lisp Objects

People cannot deal directly with Lisp objects, because the objects live inside the machine. In order to let us get at and talk about Lisp objects, Lisp provides a representation of objects in the form of printed text; this is called the *printed representation*.

Functions such as **print**, **prin1**, and **princ** take a Lisp object and send the characters of its printed representation to a stream. These functions (and the internal functions they call) are known as the *printer*. The **read** function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds a corresponding object and returns it; **read** and its subfunctions are known as the *reader*. See the section "Introduction to Streams".

The printed representation of an object depends on its type. For descriptions of how different Lisp objects are printed, See the section "What the Printer Produces".

Effects of Slashification on Printing

Printing is done either with or without *slashification*. The unslashified version is nicer looking, but **zl:read** cannot handle it properly. The slashified version, however, is carefully set up so that **zl:read** is able to read it in.

The primary effects of slashification are:

- Special characters used with other than their normal meanings (for example, a parenthesis appearing in the name of a symbol) are preceded by slashes or cause the name of the symbol to be enclosed in vertical bars.
- Symbols that are not from the current package are printed out with their package prefixes. (A package prefix looks like a symbol followed by a colon).

What the Printer Produces

The printed representation of lists is documented elsewhere. See the section "Printed Representation of Lists".

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*) "")**.

Printed Representation of Common Lisp Character Objects

For Common Lisp, character objects always print as #*char*.

Printed Representation of Strings

If slashification is off, the printed representation of a string is simply the successive characters of the string. If slashification is on, the string is printed between double quotes, and any characters inside the string that need to be preceded by slashes are. Normally these are just double-quote and slash. Compatibly with Maclisp, carriage return is *not* ignored inside strings and vertical bars.

Printed Representation of Instances

If the instance has a method for the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure, and whether slashification is enabled. The object should print a suitable printed representation on the stream. (See the section "Flavors". Instances are discussed there.) See the section "Printed Representation of Miscellaneous Data Types". Most such objects print as described there, except with additional information such as a name. Some objects print only their name when slashification is not in effect (when **princ**ed).

Printed Representation of Arrays That Are Named Structures

If the array has a named structure symbol with a **named-structure-invoke** property that is the name of a function, then that function is called on five arguments:

- The symbol **:print-self**
- The object itself
- The stream to print to
- The current *depth* of list structure
- Whether slashification is enabled

A suitable printed representation should be sent to the stream. This allows you to define your own printed representation for the array's named structures. See the section "Named Structures". If the named structure symbol does not have a **named-structure-invoke** property, the printed representation is like that for miscellaneous data types: a number sign and a less-than sign ("<"), the named structure symbol, the numerical address of the array, and a greater-than sign (">").

Printed Representation of Arrays That Are Not Named Structures

The printed representation of an array that is not a named structure contains the following elements, in order:

- A number sign and a less-than sign ("**<**")
- The "**art-**" symbol for the array type
- The dimensions of the array, separated by hyphens
- A space, the machine address of the array, and a greater-than sign ("**>**")

Printed Representation of Miscellaneous Data Types

For a miscellaneous data type, the printed representation starts with a number sign and a less-than sign, the "**dtp-**" symbol for this data type, a space, and the octal machine address of the object. Then, if the object is a microcoded function, compiled function, or stack group, its name is printed. Finally, a greater-than sign is printed.

Including the machine address in the printed representation makes it possible to tell two objects of this kind apart without explicitly calling **eq** on them. This can be very useful during debugging. It is important to know that if garbage collection is turned on, objects are occasionally moved, and therefore their octal machine addresses are changed. It is best to shut off garbage collection temporarily when depending on these numbers.

None of the printed representations beginning with a number sign can be read back in, nor, in general, can anything produced by instances and named structures. See the section "What the Reader Recognizes". This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. But by setting the ***print-readably*** variable, you can make sure that what you are printing can indeed be read with the reader.

print-readably A boolean that signals an error if the object to be printed is not in a form that the reader will accept.

sys:printing-random-object *(object stream* &rest *either of: :no-pointer or :typep)* &body *body*
The vast majority of objects that define **sys:print-self** methods have much in common. This macro is provided for convenience, so that users do not have to write out that repetitious code. It is also the preferred interface to **sci:*print-readably***.

Controlling the Printed Representation of an Object

If you want to control the printed representation of an object, usually you make the object an array that is a named structure, or an instance of a flavor. See the section "Named Structures". See the section "Flavors". Occasionally, however, you might want to get control over all printing of objects in order to change in some way how they are printed. The best way to do this is to customize the behavior of **si:print-object**, which is the main internal function of the printer. All the printing functions, such as **print** and **princ**, as well as **format**, go through this function. The way to customize it is by using the "advice" facility. See the special form **advise**.

Output Functions

For producing formatted output, see the section "Formatted Output".

Functions That Print or Write Output

The following functions take an optional argument called *output-stream*, which is where to send the output. If unsupplied or **nil**, *output-stream* defaults to the value of ***standard-output***. If it is **t**, the value of ***terminal-io*** is used (that is, the interactive terminal). See the section "Introduction to Streams".

write *object* &key *:stream :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*
The printed representation of *object* is written to the output stream specified by **:stream**.

prin1 *object* &optional *output-stream*
Outputs the printed representation of *object* to *stream*, with slashification. Roughly speaking, the output from **prin1** is suitable for input to the function **zl:read**.

zl:prin1-then-space *object* &optional *output-stream*
Like **prin1** except that output is followed by a space.

prin1-to-string *object*
The object is printed as if by **prin1**, and the characters that would be output are made into a string, which is returned.

print *object* &optional *output-stream*
Like **prin1** except that output is preceded by a Newline and followed by a space.

princ *object* &optional *output-stream*
Like **prin1** except that the output is not slashified. The general rule is that output from **princ** is intended to look good to people, while output from **prin1** is intended to be acceptable to the function **read**.

princ-to-string *object*
The object is printed as if by **princ**, and the characters that would be output are made into a string, which is returned.

pprint *object* &optional *output-stream*
The printed representation of *object* is written to the *output-stream* using the pretty printer.

write-byte *integer binary-output-stream*
Writes one byte, the value of *integer* to *binary-output-stream*.

write-char *character* &optional *output-stream*
Outputs *character* as a printing character to *output-stream*, and returns *character* as a character object.

write-line *string* &optional *output-stream* &key *(:start* **0***) :end*
Writes the characters of the specified substring of *string* to *output-stream*, followed by a newline.

write-string *string* &optional *output-stream* &key *(:start* **0***) :end*
Writes the characters of the specified substring of *string* to *output-stream*, without a following newline.

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
The object is printed as if by **write**, and the characters that would be output are made into a string, which is returned.

si:print-object *exp i-prindepth slashify-p stream* &optional *(which-operations* **(funcall si:stream ':which-operations)***) recursive*
Outputs the printed representation of *object* to *stream*, as modified by *prindepth* and *slashify-p*.

si:print-list *exp i-prindepth slashify-p stream which-operations*
This is the part of the Lisp printer that prints lists. A stream's **:print** handler can call this function.

zl:tyo *char* &optional *stream*
Outputs the character *char* to *stream*.

fresh-line &optional *output-stream*
Outputs a newline only if the stream is not already at the start of a line.

terpri &optional *output-stream*
Outputs a newline to *output-stream*, and returns **nil**.

zl:terpri &optional *stream*
Outputs a carriage return character to *stream*.

clear-output &optional *output-stream*
Attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination.

finish-output &optional *output-stream*
Attempts to ensure that all output sent to *output-stream* has reached its destination, and only then returns **nil**.

force-output &optional *output-stream*
Initiates the emptying of any internal buffers, but returns **nil** without waiting for completion or acknowledgment.

In addition to the functions summarized above, the **zl:format** function is very useful for producing nicely formatted text. **zl:format** can generate a string or output to a stream. See the section "Formatted Output".

The **grindef** function is useful for formatting Lisp programs. See the special form **grindef**.

See the function **with-output-to-string**.

Print Control Variables

There are a number of print variables that affect the performance of print functions.

print-level Controls how many levels of a nested data object will be printed.

print-length Controls how many elements at a given level are printed.

print-abbreviate-quote
Provides a way to print quoted forms in their short form. It is incorporated into ***print-pretty***, so the value of ***print-pretty*** must be **nil** in order for ***print-abbreviate-quote*** to have any effect.

print-array A boolean which controls whether the contents of arrays other than strings are printed.

print-array-length
Controls the number of objects in the array that will be printed.

print-base Determines the radix in which the printer prints rational numbers (integers and ratios).

print-bit-vector-length
Controls the number of objects in the bit vector that will be printed.

print-case Controls the case in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used.

print-circle Controls whether or not the printer tries to detect cycles in the structure to be printed.

print-escape Controls whether or not the printer outputs escape characters.

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 version, which is normally printed by the printer.

print-gensym Controls whether the prefix #: is printed before symbols that have no home package.

print-integer-length
Controls the printing of **bignum**s.

print-pretty Controls the amount of whitespace output when printing an expression.

print-pretty-printer
Allows wholesale replacement of the pretty printer used by Common Lisp.

print-radix If **t**, rational numbers are printed with a radix specifier indicating what radix the printer is using.

print-readably A boolean that signals an error if the object to be printed is not in a form that the reader will accept.

print-string-length

Controls the number of string characters that will print.

print-structure-contents

Controls how structures are printed. The default is **t**, which uses the `#S` convention, printing the structure with all its slots filled in.

Functions for Formatting Lisp Code

grindef &rest *fcns* Prints the definitions of one or more functions, with indentation to make the code readable.

zl:grind-top-level *exp* &optional *si:grind-width (si:grind-real-io* **zl:standard-output***) si:grind-untyo-p (si:grind-displaced* **'si:displaced***) (terpri-p* **t***) si:grind-notify-fun (loc* **(ncons exp)***)*

Pretty-prints *exp* on *stream*, inserting up to *si:grind-width* characters per line. This is the primitive interface to the pretty-printer.

Miscellaneous Output Functions

stream-copy-until-eof *from-stream to-stream* &optional *leader-size* &key *(:include-diagrams* **t***)*

Inputs characters from *from-stream* and outputs them to *to-stream* until it reaches the end-of-file on the *from-stream*.

beep &optional *beep-type (stream* **zl:terminal-io***)*

Tries to attract the user's attention by causing an audible beep, or flashing the screen, or something similar.

sys:printing-random-object *(object stream* &rest *either of: :no-pointer or :typep)* &body *body*

The vast majority of objects that define **sys:print-self** methods have much in common. This macro is provided for convenience, so that users do not have to write out that repetitious code. It is also the preferred interface to **sci:*print-readably***.

zl:cursorpos &rest *args*

This function exists primarily for Maclisp compatibility. It performs operations related to the cursor position, such as returning the position, moving the position, or performing another cursor operation.

zl:exploden *x* Returns a list of characters (as integers) that are the characters that would be typed out by **(princ** *x***)** (that is, the unslashified printed representation of *x*).

zl:explodec *x* Returns a list of characters represented by symbols that are the characters that would be typed out by **(princ** *x***)** (that is, the unslashified printed representation of *x*).

zl:explode *x* — Returns a list of characters represented by symbols that are the characters that would be typed out by **(prin1** *x***)** (that is, the slashified printed representation of *x*).

sys:flatsize *x* — Returns the number of characters in the slashified printed representation of *x*.

sys:flatc *x* — Returns the number of characters in the unslashified printed representation of *x*.

Formatted Output

You can control the appearance of program output using the **format** function. **format** uses a control string written in a special format specifier language to control the output format.

For simple tasks in which only the most basic format specifiers are needed, **format** is easy to use and has the advantage of brevity. For more complicated tasks, the format specifier language becomes somewhat obscure and hard to read, but has the advantage of being extremely powerful.

You can also use other functions that are specially designed for complex formatting tasks. Some of these include **format:print-list** and **format-textual-list**.

A number of global control variables affect the results of **format**. See the section "Print Control Variables".

Additional tools are available for formatting Lisp code (as opposed to text and tables). See the section "Functions for Formatting Lisp Code".

format *destination control-string* &rest *args* — *Function*

Formats output as specified by *control-string* and sends it to *destination*.

control-string contains the format directives and optionally, all or part of the intended output. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

Format directives begin with a tilde (~). The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies the kind of formatting desired. For more on format directives: See the section "Format Directives".

destination specifies what **format** does with the results:

nil — A string is created that contains the output; this string is returned as the value of the call to **format**.

t — The output is sent to ***standard-output***. **format** returns **nil**.

stream — The output is sent to *stream*. **format** returns **nil**.

string The output is added to the end of *string*. *string* should have a fill-pointer. See the section "What is a Fill Pointer?". **format** returns **nil**.

format includes some extremely complicated and specialized features. It is not necessary to know all or even most of its features to use **format** efficiently. The more sophisticated features are there for use in programs with complicated formatting requirements.

Compatibility Note for format

Some format directives are named, that is, instead of a one-character directive, such as ~**A**, the directive has a name within backslashes (\), such as ~**\\date**.

The escape character for the Common Lisp reader is the backslash \, while the escape character for the Zetalisp reader is the slash /. In order to output a backslash character in Common Lisp it is necessary to double the character when typing it.

Therefore, the name of such a directive in a Common Lisp environment is ~**\\date**, while the name in a Zetalisp environment is ~**\date**. If you are using any of the named directives, it is very important that you keep track of the package in which you are evaluating or compiling such code.

Format Directives

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (`:`) and at-sign (`@`) modifiers, and a single character indicating the kind of formatting desired. The alphabetic case of this character is ignored. The prefix parameters are generally signed integers. Examples of directives:

```
"~S"        ; This is an S directive with no parameters or modifiers.
"~3,4:@s"   ; This is an S directive with two parameters, 3 and 4,
            ;    and both the colon and at-sign flags.
"~,4S"      ; The first prefix parameter is omitted and takes
            ;    on its default value, while the second is 4.
```

Sometimes a prefix parameter is used to specify a character, such as the padding character in a right- or left-justifying operation. In this case a single quote (`'`) followed by the desired character can be used as a prefix parameter. For example, you can use the following to print a decimal number in five columns with leading zeros.

```
"~5,'0d"  instead of "~5,48d"
```

In place of a prefix parameter to a directive, you can put the letter **V**, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it does not have to be. This feature allows variable column-widths and the like. Also, you can use the character `#` in place of a parameter; it represents the number of arguments remaining to be processed.

Formatting Examples

Here are some examples of how **format** is used. They range from the very simple to some of the more complex operations. More information about these examples can be found in the section that discusses the appropriate format directive.

```
(format nil "Simple output") => "Simple output"

(setq x 5)
(format nil "Print a decimal number: ~D" x) =>
"Print a decimal number: 5"
(format nil "Print a decimal number and some space: ~5D." x) =>
"Print a decimal number and some space:     5."

(setq y "elephant")
(format nil "Print a symbol: ~A!" y) =>
"Print a symbol: elephant!"

(format nil "Print out part of this string, a carriage return ~% ~
   and then an odd character: ~:C"
   (set-char-bit #\D :control t)) =>
"Print out part of this string, a carriage return
 and then an odd character: Control-D"

(setq n 3)
(format nil "Pluralize based on the value of n.~
   ~%~D item~:P found." n) =>
"Pluralize based on the value of n.
3 items found."

(format nil "Select the verb based on the value of n.~
   ~%Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n) =>
"Select the verb based on the value of n.
Here are three puppies."

(setq foo "Decide how to format the items based ~
   on how many there are. ~%Items:~#[ none~; ~S~; ~S and ~S~:;~
   ~@{~#[~1; and~] ~S~^,~}~].")
(format nil foo 'foo 'bar) =>
"Decide how to format the items based on how many there are.
Items: FOO and BAR."

(format nil "Print in ~'i⊂italics~⊃ or ~V⊂bold~⊃."
        '(:fix :bold :normal)) =>
"Print in italics or bold."
```

Format Directives That Operate on Lisp Objects

~A

Prints the next element from the *args* of the **format** function without slashification (as by **princ**). If you simply specify *mincol*, as in ~*mincol***A**, **format** inserts spaces on the right, if necessary, to make the column width at least *mincol*.

The format is:

~*mincol*,*colinc*,*minpad*,*padchar*:@**A**

It expects its argument to be a Lisp object.

mincol	The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.
colinc	Padding characters are inserted *colinc* characters at a time. The default is **1**.
minpad	The minimum number of copies of *padchar* to use. The default is **0**.
padchar	The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.
:	Prints **( )** if the element is **nil**; this is useful when printing something that is always supposed to be a list.
@	Inserts *padchar* on the left rather than the right.

Examples:

```
(setq arg '|elephant|) => |elephant|
(format nil "Look at the ~A!" arg) => "Look at the elephant!"
(format nil "Look at the ~12A!" arg) => "Look at the elephant    !"
(format nil "Look at the ~12@A!" arg) =>
"Look at the     elephant!"
(format nil "Look at the ~12,3,,'!A" arg) =>
"Look at the elephant!!!!"

(setq arg '()) => NIL
(format nil "Look at the ~10A!" arg) => "Look at the NIL       !"
(format nil "Look at the ~10:A!" arg) => "Look at the ()        !"
```

~S

Prints the next element from the *args* of the **format** function with slashification (as by **prin1**). If you simply specify *mincol*, as in ~*mincol***S**, **format** inserts spaces on the right, if necessary, to make the column width at least *mincol*. For an explanation of slashification: See the section "What the Printer Produces".

The format is:

~mincol,colinc,minpad,padchar:@**S**

It expects its argument to be a Lisp object.

mincol	The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.
colinc	Padding characters are inserted *colinc* characters at a time. The default is **1**.
minpad	The minimum number of copies of *padchar* to use. The default is **0**.
padchar	The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.
:	Prints **( )** if the element is **nil**; this is useful when printing something that is always supposed to be a list.
@	Inserts *padchar* on the left rather than the right.

Examples:

```
(setq arg '#\b)
(format nil "Print a character: ~S" arg) => "Print a character: #\\b"

(setq arg '|elephant|) => |elephant|
(format nil "Look at the ~S!" arg) => "Look at the |elephant|!"
(format nil "Look at the ~12S!" arg) => "Look at the |elephant|  !"
(format nil "Look at the ~12@S!" arg) =>
"Look at the   |elephant|!"
(format nil "Look at the ~12,3,,'!S" arg) =>
"Look at the |elephant|!!!"

(setq arg '()) => NIL
(format nil "Look at the ~10S!" arg) => "Look at the NIL       !"
(format nil "Look at the ~10:S!" arg) => "Look at the ()      !"
```

Format Directives That Operate on Numbers

~D

Prints the next element from the *args* of the **format** function in decimal radix. The *arg* should be an integer. If it is not, it is printed in **~A** format and decimal base. **~D** never puts a decimal point after the integer. If you simply specify *mincol*, as in *~mincol***D**, **format** inserts spaces on the left, if the number requires fewer than *mincol* columns for its digits and sign. If the number does not fit in *mincol* columns, additional columns are used as needed.

The format is:

~mincol,padchar,commachar:@**D**

It expects its argument to be an integer. If the argument is not an integer, it is printed in **~A** format and decimal base. Thus, this directive can be used to print some list structure, showing all fixnums in decimal.

mincol	The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.
padchar	The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.
commachar	The character used to separate groups of three digits within the number. The default is **#:|#,|**.
:	Prints *commachar* between groups of three digits.
@	Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq arg 123456) => 123456
(format nil "~D" arg) => "123456"
(format nil "~12,'~D" arg) => "~~~~~~123456"
(format nil "~12,'~:@D" arg) => "~~~~+123,456"
(format nil "~5,'~:@D" arg) => "+123,456"

(setq arg '(31. 53. 79. 11.)) => (31 53 79 11)
(format nil "~D" arg) => "(31 53 79 11)"
(format nil "~A" arg) => "(31 53 79 11)"
```

~B

Prints the next element from the *args* of the **format** function in binary radix (base 2). **~B** never puts a decimal point after the number nor a **#b** prefix before the number. If you simply specify *mincol*, as in *~mincol***B**, **format** inserts spaces on the left, if the number requires fewer than *mincol* columns for its digits and sign. If the number does not fit in *mincol* columns, additional columns are used as needed.

The format is:

~mincol,padchar,commachar:@**B**

It expects its argument to be an integer. If the argument is not an integer, it is printed in **~A** format and binary base. Thus, this directive can be used to print some list structure, showing all integers in binary.

mincol — The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.

padchar — The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

commachar — The character used to separate groups of three digits within the number. The default is **#:|#,|**.

: — Prints *commachar* between groups of three digits.

@ — Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq arg 245) => 245
(format nil "~B" arg) => "11110101"
(format nil "~12,'~B" arg) => "~~~~11110101"
(format nil "~12,'~:@B" arg) => "~+11,110,101"
(format nil "~5,'~:@B" arg) => "+11,110,101"

(setq arg '(31. 53. 79. 11.)) => (31 53 79 11)
(format nil "~B" arg) => "(11111 110101 1001111 1011)"
(format nil "~A" arg) => "(31 53 79 11)"
```

~O

Prints the next element from the *args* of the **format** function in octal radix (base 8). **~O** never puts a decimal point after the number nor a **#o** prefix before the number. If you simply specify *mincol*, as in *~mincol***O**, **format** inserts spaces on the left, if the number requires fewer than *mincol* columns for its digits and sign. If the number does not fit in *mincol* columns, additional columns are used as needed.

The format is:

~mincol,padchar,commachar:@**O**

It expects its argument to be an integer. If the argument is not an integer, it is printed in **~A** format and octal base. Thus, this directive can be used to print some list structure, showing all integers in octal.

mincol — The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.

padchar — The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

commachar — The character used to separate groups of three digits within the number. The default is **#:|#,|**.

: — Prints *commachar* between groups of three digits.

@ — Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq arg 245) => 245
(format nil "~O" arg) => "365"
(format nil "~12,'~O" arg) => "~~~~~~~~~365"
(format nil "~12,'~:@O" arg) => "~~~~~~~~+365"
(format nil "~2,'~:@O" arg) => "+365"

(setq arg '(31. 53. 79. 11.)) => (31 53 79 11)
(format nil "~O" arg) => "(37 65 117 13)"
(format nil "~A" arg) => "(31 53 79 11)"
```

~X

Prints the next element from the *args* of the **format** function in hexadecimal radix (base 16). **~X** never puts a decimal point after the number nor a **#x** prefix before the number. If you simply specify *mincol*, as in *~mincol***X**, **format** inserts spaces on the left, if the number requires fewer than *mincol* columns for its digits and sign. If the number does not fit in *mincol* columns, additional columns are used as needed.

The format is:

*~mincol,padchar,commachar***:@X**

It expects its argument to be an integer. If the argument is not an integer, it is printed in **~A** format and hexadecimal base. Thus, this directive can be used to print some list structure, showing all integers in hexadecimal.

mincol — The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.

padchar — The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

commachar — The character used to separate groups of three digits within the number. The default is **#:|#,|**.

: — Prints *commachar* between groups of three digits.

@ — Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq arg 1234567890) => 1234567890
(format nil "~X" arg) => "499602D2"
(format nil "~12,'~X" arg) => "~~~~499602D"
(format nil "~12,'~:@X" arg) => "~+49,960,2D2"
(format nil "~5,'~@X" arg) => "+499602D2"

(setq arg '(31. 53. 79. 11.)) => (31 53 79 11)
(format nil "~X" arg) => "(1F 35 4F B)"
(format nil "~A" arg) => "(31 53 79 11)"
```

Zetalisp Note: The ~**X** directive for **zl:format** prints a space. If you want to output multiple spaces, you can use either a numerical prefix or the **V** prefix.

For example:

```
(zl:format nil "~15X") => "               "

(setq n 15)
(zl:format nil "~VX" n) => "               "
```

~E

Prints the next element from the *args* of the **format** function in exponential notation. Given the prefixes *w*, *d* and *e*, the argument prints as *w* characters wide, filled as follows: appropriate padding on the left, a possible sign, a sequence of digits containing an embedded decimal point that represents the fractional part of the argument, an exponent character, a sign, and a sequence of digits representing the power of ten by which the fraction must be multiplied to properly represent the rounded value of the argument.

If *w*, *d* and *e* are omitted, the argument is printed using ordinary free-format exponential notation (such as **prin1** uses on very small or very large numbers).

The format is:

~*w*,*d*,*e*,*k*,*overflowchar*,*padchar*,*exponentchar*@**E**

It expects its argument to be a floating-point number. If the argument is a rational number, it is coerced to be a floating-point number. If the argument is a complex number or a non-numeric object, it is printed using the format directive ~*w***D**. This prints the argument as a decimal integer with a field width of *w*.

w

The width of the output field. If *w* is present, and the argument is printable within the specified width, then the argument is printed with enough padding on the left to fill *w* characters.

If *w* is present and the argument is not printable within the specified width, one of two alternatives may occur, depending on whether *overflowchar* is specified. If *overflowchar* is speci-

fied, *w* copies of that character are printed. If *overflowchar* is not specified, the argument is printed using as many more characters as are needed.

If *w* is not present, the field width is variable. The argument is printed with no leading pad characters.

d — The number of digits to print after the decimal point. The number of digits printed depends on the scale factor *k*. If *k* is zero, *d* digits are printed after the decimal point, and a zero appears before the decimal point if the total field width permits.

If *k* is either not present or positive, it must be less than *d*+2. In this case, *k* significant digits are printed before the decimal point and *d*-*k*+1 digits are printed after the decimal point.

If *k* is negative, it must be greater than -*d*. In this case, a single 0 appears before the decimal point if the total field width permits, and *k* zeros followed by *d*+*k* significant digits are printed after the decimal point.

e — The number of digits to print for the exponent. This is the power of ten by which the printed fraction must be multiplied to represent the value of the argument. If *e* is not present, the exponent is printed using the smallest number of digits necessary to represent its value.

k — A scale factor that interacts with *d* to determine the format of the fractional part. The default is **1**.

overflowchar — A character printed in place of the argument when it is impossible to fit the argument within the specified format.

padchar — The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

exponentchar — The character printed to flag the start of the exponent part of the argument. The default is \#e.

@ — Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq arg 123.456) => 123.456
(format nil "~E" arg) => "123.456"
(format nil "~15,6,2,,'%,'~,'E@E" arg) => "~~+1.234560E+02"
(format nil "~15,6,2,5,'%,'~,'E@E" arg) => "~~+12345.60E-02"
(format nil "~15,6,2,-2,'%,'~,'E@E" arg) => "~~+0.001235E+05"
(format nil "~10,6,2,,'%,'~,'E@E" arg) => "%%%%%%%%%%"
(format nil "~15,8,2,,'%,'~,'E@E" arg) => "+1.23456000E+02"
(format nil "~15,4,2,,'%,'~,'E@E" arg) => "~~~~+1.2346E+02"
```

Zetalisp Note: The ~**E** directive for **zl:format** has different prefix notation.

The format is:

~*n*@**E**

It expects its argument to be a floating-point number. If the element is not a number, it is printed in ~**A** format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired.

n ~*n***E** rounds the element to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed.

```
(setq arg 123.456) => 123.456
(zl:format nil "~3E" 5.) => "5.0e0"
(zl:format nil "~E" arg) => "1.23456e2"
(zl:format nil "~10E" arg) => "1.23456e2"
(zl:format nil "~3E" arg) => "1.23e2"
```

~F

Prints the next element from the *args* of the **format** function in floating-point format. Given the prefixes *w* and *d*, the argument prints as *w* characters wide, filled as follows: appropriate padding on the left, a possible sign, then a sequence of digits containing an embedded decimal point. The sequence of digits represents the value of the argument multiplied by 10^k, rounded to *d* digits beyond the decimal point.

If *w* and *d* are omitted, the argument is printed using ordinary free-format notation (such as **prin1** uses on smaller numbers).

The format is:

~*w*,*d*,*k*,*overflowchar*,*padchar*@**F**

It expects its argument to be a floating-point number. If the argument is a rational number, then it is coerced to be a floating-point number. If the argument is a complex number or a non-numeric object, it is printed using the format directive ~*w***D**. This prints the argument as a decimal integer with a field width of *w*.

w The width of the output field. If *w* is present, and the argument is printable within the specified width, then the argument is printed with enough padding on the left to fill *w* characters.

If *w* is present and the argument is not printable within the specified width, one of two alternatives may occur, depending on whether *overflowchar* is specified. If *overflowchar* is specified, *w* copies of that character are printed. If *overflowchar* is not specified, the argument is printed using as many more characters as are needed.

If *w* is not present, the field width is variable. The argument is printed with no leading pad characters.

d — The number of digits to print after the decimal point. If *d* is omitted, as many digits as possible are printed after the decimal point, subject to the width *w* and the constraint that no trailing zeroes may appear in the fraction.

k — A scale factor used to adjust the argument by factors of ten. The argument is multiplied by 10^k. The default is **0**.

overflowchar — A character printed in place of the argument when it is impossible to fit the argument within the specified format.

padchar — The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

@ — Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq arg 123.456) => 123.456
(format nil "~F" arg) => "123.456"
(format nil "~12,4,,'%,'~F" arg) => "~~~~123.4560"
(format nil "~12,4,2,'%,'~F" arg) => "~~12345.6001"
(format nil "~8,4,,'%,'~F" arg) => "123.4560"
(format nil "~12,6,,'%,'~F" arg) => "~~123.456000"
(format nil "~10,2,,'%,'~F" arg) => "~~~123.46"
```

Zetalisp Note: The ~**F** directive for **zl:format** has different prefix notation.

The format is:

~*n*@**F**

It expects its argument to be a floating-point number. If the element is not a number, it is printed in ~**A** format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired.

n — ~*n***F** rounds the element to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed. If the magnitude of the element is too large or too small, it is printed in exponential notation.

```
(setq arg 123.456) => 123.456
(zl:format nil "~3F" 5) => "5.0"
(zl:format nil "~F" arg) => "123.456"
(zl:format nil "~4F" arg) => "123.5"
(zl:format nil "~8F" arg) => "123.456"
(zl:format nil "~3F" 1e10)  =>  "1.0e10"
```

~$

Prints the next element from the *args* of the **format** function in fixed-format floating-point notation for dollars and cents.

The format is:

~rdig,ldig,field,padchar:@**$**

It expects its argument to be a floating-point number. If the argument is not a floating-point number, it is printed using the format directive *~field***D**. This prints the argument as a decimal integer with a field width of *field*.

rdig	The number of digits to be printed after the decimal point. The default is **2**.
ldig	The minimum number of digits to be printed before the decimal point. The default is **1**.
field	The full width of the field to print in. The field is padded to the left with *padchar* if the number of characters output is less than *field*. The default is **0**.
padchar	The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.
:	Prints the sign character at the beginning of the field, before the padding, rather than just to the left of the number.
@	Always prints the sign of the number. The default is to print it only if the number is negative.

Examples:

```
(setq n 1459.32)
(format nil "The amount is $~$" n) =>
"The amount is $1459.32"
(format nil "The amount is $~,2,16:@$" n) =>
"The amount is $+        1459.32"
(format nil "The amount is $~,,16,'x$" n) =>
"The amount is $xxxxxxxx+1459.32"
```

~G

Prints the next element from the *args* of the **format** function in either fixed-format or exponential floating-point notation using either the **~E** or the **~F** format directive.

Whether **~E** or **~F** will be used is determined by the magnitude of the argument and the number of digits to print after the decimal point. Let n be an integer such that $10^{n-1} \leq arg < 10^n$. If *arg* is zero, let n be 0. Let dd be $d - n$. If d is omitted, let q be the number of digits needed to print *arg* with no loss of information and without leading or trailing zeros. Then let d be **(max** q **(min** n **7))**. If $0 \leq dd < d$, **~F** is used, otherwise the **~E** format directive is used.

When the **~F** format directive is used, the format is:

~ww,dd,,overflowchar,padchar@**F***~ee*@**T**

where *ee* is $e + 2$ (or 4 if e is omitted) and *ww* is $w - ee$. Note that the scale factor k is not passed to **~F**. Also note that *ee* spaces are printed after the number. The @ modifier is passed to **~F** only if one was specified in the **~G** directive.

When the **~E** format directive is used, the format is:

~w,d,e,k,overflowchar,padchar,exponentchar@**E**

The @ modifier is passed to **~E** only if one was specified in the **~G** directive.

The format for the **~G** format directive is:

~w,d,e,k,overflowchar,padchar,exponentchar@**G**

It expects its argument to be a floating-point number. If the argument is a rational number, it is coerced to be a floating-point number. If the argument is a complex number or a non-numeric object, it is printed using the format directive *~w***D**. This prints the argument as a decimal integer with a field width of w.

For an explanation of the arguments, see the appropriate format directive. See the section "**~F**". See the section "**~E**".

Examples:

```
(setq arg1 123.456) => 123.456
(setq arg2 1234567890.123) => 1.234568e9
(format nil "~G" arg1) => "123.456    "
(format nil "~G" arg2) => "1.234568e+9"
(format nil "~15,6,2,,'%,'~,'E@G" arg1) => "~~~+123.456    "
(format nil "~15,6,2,,'%,'~,'E@G" arg2) => "~~+1.234568E+09"
(format nil "~15,6,2,5,'%,'~,'E@G" arg1) => "~~~+123.456    "
(format nil "~15,6,2,5,'%,'~,'E@G" arg2) => "~~+12345.68E+05"
```

Zetalisp Note: The **~G** directive for **zl:format** performs a different function.

In Zetalisp, **~G** "goes to" the nth argument. Directives following this one correspond to the sequence of arguments following the argument that is the target of **~G**. Inside a **~{** directive, the "goto" is relative to the list of arguments being processed by the iteration. This is an absolute goto. For a relative goto: See the section "**~***".

The format is:

~*n*G

It takes no arguments.

n ~*n*G branches to the *n*th element in the argument list. For example, ~0G goes back to the first argument in the *args* of the **format** function.

~R

Prints the next element from the *args* of the **format** function in a specified radix, or as a number in special format. When *radix* is specified, prints the argument in *radix*. If there are no prefixes, prints the argument as a cardinal English number (for example, four).

The format is:

~*radix*,*mincol*,*padchar*,*commachar*:@R

It expects its argument to be an integer.

radix The base to use for printing the argument.

mincol The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.

padchar The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

commachar The character used to separate groups of three digits within the number. The default is **#:|#,|**.

: Prints the sign character at the beginning of the field, before the padding, rather than just to the left of the number. When no prefixes are given, prints as an ordinal number (for example, fourth).

@ Always prints the sign of the number. The default is to print it only if the number is negative. When no prefixes are given, prints as a Roman numeral (for example, IV).

:@ Performs both the : and @ operations. When no prefixes are given, prints as an old Roman numeral (for example, IIII).

Examples:

```
(format nil "~R" 14) => "fourteen"
(format nil "~@R" 14) => "XIV"
(format nil "~3R" 14) => "112"
(format nil "~8,5,'~R" 14) => "~~~16"
(format nil "~8,5,'~:@R" 1400) => "+2,570"
```

~P

Prints the non-directive text immediately preceding **~P** in the *control-string* either singular or plural depending on whether the next element from the *args* of the **format** function is 1 or not. If the next element in the *args* of the **format** function is not **1**, a lowercase "s" is output.

The format is:

~:@P

It expects its argument to be an integer.

:	Prints a possible "s" after backing up one element in the argument list. That is, prints a lowercase "s" if the previous argument is not 1.
@	Prints a lowercase "y" if the next argument is 1 or "ies" if it is not.
:@	Prints a possible "y" after backing up one element in the argument list. That is, prints a lowercase "y" if the previous element in the list of arguments is 1 or "ies" if it is not.

Examples:

```
(format nil "We have ~D dog~:P" 1) => "We have 1 dog"
(format nil "We have ~D dog~:P" 5) => "We have 5 dogs"
(format nil "We have ~D pupp~:@P" 1) => "We have 1 puppy"
(format nil "We have ~D pupp~:@P" 5) => "We have 5 puppies"
```

Format Directives That Operate on Characters

~C

Prints the next element from the *args* of the **format** function as a character object. The argument is treated as a keyboard character and thus can contain extra modifier bits. The constants **char-control-bit**, **char-meta-bit**, **char-hyper-bit**, **char-super-bit**, and **char-bits** return the modifier bits for characters. The modifier bits are printed first, represented as appropriate prefixes: `c-` for Control, `m-` for Meta, `c-m-` for Control plus Meta, `h-` for Hyper, `s-` for Super. If the character is not a keyboard character but a mouse character, it is printed as `Mouse-`, the name of the button, `-`, and the number of clicks.

The format is:

~:@C

It expects its argument to be a character object.

: Spells out the names of the modifier bits (for example, Control-Meta-F), and represent non-printing characters by their names (for example, RETURN) rather than as objects.

@ Prints the character in such a way that the Lisp reader can understand it, using "#/" or "#\".

:@ Prints the colon-only format, and if the SYMBOL or another special key is required to type it, this fact is mentioned (for example, SYMBOL-1). This is the format used in prompt messages, for instance, to tell the user about a key he or she is expected to press.

Examples:

Character	~C	~:C	~@C	~:@C
#\a	a	a	#\\a	a
#\Return	Does it	Return	#\\Return	Return
#\Mouse-L	Mouse-L	Mouse-Left	#\\Mouse-Left	Mouse-Left
#\Control-D	c-D	Control-D	#\\c-D	Control-D
#\Center-Dot	·	Center-Dot	#\\Center-Dot	Center-Dot (Symbol-')

```
(format nil "The character ~:@C is normal." #\F) =>
"The character F is normal."
(format nil "The character ~:@C is strange." #o0) =>
"The character Center-Dot (Symbol-') is strange."
(format nil "Type ~:C to ~A"
  (set-char-bit #\D :control t)
  "delete all your files") =>
"Type Control-D to delete all your files"
```

~◊

Prints the name of the next element from the *args* of the **format** function inside a lozenge. The **~C** directive does this with some characters, but ~◊ does it with all of them. If the stream does not support drawing a lozenge, it will enclose the character within angle brackets. This **format** directive is a Symbolics extension to Common Lisp.

The format is:

~◊

It expects its argument to be a character object.

Examples:

```
(format nil "Press ~◊ when ready." #\Return) =>
"Press <Return> when ready."
(format nil "Type ~◊ to continue." #\y) =>
"Type <y> to continue."
```

Format Directives That Operate on Whitespace

~T

Spaces over to a given column in units of characters. If you simply specify *colnum*, as in ~*colnum***T**, output *colnum* spaces. If you specify *colnum* and *colinc*, as in ~*colnum*,*colinc***T**, prints enough spaces to move the cursor to column *colnum*, or if the cursor is already past *colnum*, output enough spaces to move the cursor to the next tab stop beyond *colnum*.

Note: This operation works properly only on streams that support the **:read-cursorpos** and **:set-cursorpos** stream operations. On other streams, any ~**T** operation simply outputs two spaces.

The format is:

~*colnum*,*colinc***:@T**

It takes no arguments.

colnum	Outputs sufficient spaces to move the cursor to *colnum*. The default is **1**.
colinc	If the cursor is at or beyond *colnum*, output the smallest positive number of spaces necessary to move the cursor to a column that is a multiple of *colinc*. If you wish the cursor not to move if you are beyond *colnum*, set *colinc* to **0**. The default is **1**.
:	Outputs spaces in units of pixels.
@	Performs relative tabulation. ~*colnum*,*colinc***@T** outputs *colnum* spaces, then outputss the smallest positive number of spaces necessary to move the cursor to a column that is a multiple of *colinc*. If the current output column cannot be determined, *colinc* is ignored and exactly *colnum* spaces are output.

Examples:

```
(format nil "!~3T!") => "!   !"
(format nil "!~5,8T!") => "!    !"
(format nil "!~5,8@T!") => "!       !"
(format nil "!~1,0T!") => "!!"
(format nil "!~20,0T!") => "!                   !"
```

~|

Outputs a page separator character (**#:|#PAGE|**). ~*n*| outputs *n* page characters.

The format is:

~*n*:|

It takes no arguments.

: If the output stream supports the **:clear-screen** operation this directive clears the screen; otherwise it outputs page separator character(s) as if no **:** modifier were present.

Examples:

```
(format t
  "This is the end of one page. ~| And the start of another.") =>
This is the end of one page.<PAGE>And the start of another.
NIL
```

~%

Outputs a #\Newline character (carriage return plus line feed). ~*n*% outputs *n* newlines. Simply putting a Newline in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

The format is:

~*n*%

It takes no arguments.

Examples:

```
(format nil
  "This is the end of one line. ~2% And the start of another.") =>
"This is the end of one line.

 And the start of another."
```

~&

Performs a **fresh-line** operation on the output stream. Unless the stream knows that it is already at the beginning of a line, this outputs a #\Newline character. ~*n*& does a **fresh-line** operation and then outputs *n-1* more #\Newline characters.

The format is:

~*n*&

It takes no arguments.

Examples:

```
(format t
   "This is the start of a line.") => This is the start of a line.
NIL
(format t "~&This is the start of a line.") =>
This is the start of a line.
NIL
```

~<Newline>

Ignores the Newline character and any whitespace at the beginning of the next line. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

The format is:

~:@<Newline>

It takes no arguments.

:	Preserves the following whitespace.
@	Preserves the following newline.
:@	Preserves both the following Newline and whitespace. This modifier instruction is a Symbolics extension to Common Lisp.

Examples:

```
(format nil "This is the ~
 start of a line.") => "This is the start of a line."
(format nil "This is the ~:
 start of a line.") => "This is the  start of a line."
(format nil "This is the ~@
 start of a line.") => "This is the
start of a line."
(format nil "This is the ~:@
 start of a line.") => "This is the
 start of a line."
```

Special Purpose Format Directives

~~

Outputs a tilde. Since a tilde introduces a directive, ~~ must be used when you want to output a tilde character. *~n~* outputs *n* tildes.

The format is:

~n~

It takes no arguments.

Examples:

```
(format nil "~~Hello~~") => "~Hello~"
```

~*

Ignores the next element in the *args* of the **format** function. This is a "relative branch". ~*n** ignores the next *n* arguments. ~*n*@** branches to the *n*th argument (0 is the first). This is an "absolute branch". Within the ~{ format directive, the branch (in either direction) is relative to the list of arguments being processed by the iteration.

The format is:

~*n*:@*

It takes no arguments.

:	"Ignores backwards"; that is, backs up in the list of arguments so that the argument last processed processed again. ~*n*:* backs up *n* arguments.
@	Goes back to the first argument in the *args* of the **format** function. Directives after a ~*n*@* take sequential arguments after the one that is the target of the branch.

Examples:

```
(format nil
        "Pick the second arg and print it: ~* ~S"
        'foo 'bar) =>
"Pick the second arg and print it:  BAR"
(format nil
        "Print the first arg then print it again: ~S~:* ~S"
        'foo 'bar) =>
"Print the first arg then print it again: FOO FOO"
(format nil
        "Print the first arg, the second arg, then print the ~@
        first again: ~S ~S~@* ~S" 'foo 'bar) =>
"Print the first arg, the second arg, then print the
first again: FOO BAR FOO"
```

~^

Terminates the immediately enclosing ~{ or ~< format directives if no more arguments remain to be processed. If there is no such enclosing directive, terminates the entire formatting operation. In the ~< case, the formatting is performed, but no more segments are processed before doing the justification. The ~^ should appear only at the beginning of a ~< formatting clause, because it aborts the entire clause. ~^ can appear anywhere in a ~{ formatting clause.

If ~^ is used within a **~:{** clause, it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use **~:^**.

When prefixes are used, not all of the prefix parameters should be constants; at least one of them should be a **#** or a **V** parameter.

The format is:

~*n*,*m*,*p***:^**

It takes no arguments.

n	Terminates the loop only if *n* is zero.
n,*m*	Terminates the loop only if $n = m$.
n,*m*,*p*	Terminates the loop only if $n \leq m \leq p$.
:	Within an **~:{...~}** construct, terminates the entire iteration process.

Examples:

```
(setq donestr "Done.~^  ~D warning~:P.~^  ~D error~:P.")
(format nil donestr) => "Done."
(format nil donestr 3) => "Done.  3 warnings."
(format nil donestr 1 5) => "Done.  1 warning.  5 errors."
```

~Q

Escapes to arbitrary user-supplied code. Use this directive when you want to call a function, but do not want to output its returned value. *arg* is called as a function; its arguments are the prefix parameters to **~Q**, if any. *args* can also be passed to the function by using the **V** prefix parameter. Note that if you use the **V** prefix parameter, the **V** is processed before the **Q**, so the arguments are reversed. The called function can output to ***standard-output*** and can look at the variables **format:colon-flag** and **format:atsign-flag**, which are **t** or **nil**, to reflect the **:** and **@** modifiers of the **~Q**. The value return by the called function is discarded, so the only way output may be seen is if it is a side effect. This **format** directive is a Symbolics extension to Common Lisp.

The format is:

~*n***Q**

It expects its argument to be a function.

Examples:

```
(format t "~VQ" 1 'tan)
(format t "~1Q" 'tan)
```

are equivalent to saying

```
(funcall tan 1)
```

and discarding the value.

~?

Substitutes the next element in *args* of the **format** function as *control-string*, and the element after it as the list of arguments. *control-string* is processed as the new format control string, with the elements of the list following it as the corresponding arguments. The processing of the format string containing **~?** resumes when the processing of **~?**'s string is finished. This directive allows nested *control-strings*.

The format is:

~@?

It expects its argument to be a string followed by a list.

@	The next element in *args* must be a string; it is processed as part of the main format string, as if it were substituted for the **~@?** directive.

Examples:

```
(format nil "~? ~D" "<~A ~D>" '("Myname" 50.) 7) => "<Myname 50> 7"
(format nil "~@? ~D" "<~A ~D>" "Myname" 50. 7) => "<Myname 50> 7"
```

~[*str***~]**

Provides a set of alternative control strings. The alternatives (called *clauses*) are separated by **~;** and the construct is terminated by **~]**. **~]** is only used in this construct. **~;** is also used as a separator in the justification (**~<**) construct but is used in no other construct.

For example:

```
"~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
   ~;Tiger ~]kitty"
```

Where *arg* is the next element from the *args* of the **format** function, the *arg*th alternative is selected; **0** selects the first. If a prefix parameter is given (that is, **~***n***[**), then the parameter is used instead of an argument (this is useful only if the parameter is "#", which dispatches on the number of arguments left to be processed). If *arg* is out of range, no alternative is selected. After the selected alternative has been processed, the control string continues beyond the **~]**.

The format is:

~:@[*str0***~;***str1***~;...~;***strn***~:;***default***~]**

It expects its argument to be an integer.

~;	Separates the clauses.

~:;default The default clause. If the *last* **~;** used to separate clauses is instead **~:;**, then the last clause is an "else" clause, which is processed if no other clause is selected.

~[~a1,b1,...;str1~a2,b2,...;str2...~] Matches tags to the clause that follows them. The prefixes to each **~;** are numeric tags for the clause that follows it. Whichever clause has a tag matching the argument is processed.

~[~a1,a2,b1,b2,...:;str1...] Matches a range of values, *a1* through *a2* (inclusive), *b1* through *b2*, and so on, to the clause that follows them. Whichever clause matches the argument within its range of values is processed. **~:;** with no parameters can be used at the end to denote a default clause.

~:[false~;true~] Selects the *false* control string if *arg* is **nil**, and selects the *true* control string otherwise.

~@[true~] Tests the argument. If the argument is not **nil**, then it is not used up, but is the next one to be processed. If the argument is **nil**, then it is discarded, and the clause is not processed.

Dispatches the string on the number of arguments left. This is useful, for example, in dealing with English conventions for printing lists. See the function **format-textual-list**.

Examples:

```
(setq arg1 0)
(setq arg2 7)
(format nil "We have a ~[Siamese ~;Manx ~;Persian ~;Tiger ~
   ~:;Bad ~]kitty." arg1) => "We have a Siamese kitty."
(format nil "We have a ~[Siamese ~;Manx ~;Persian ~;Tiger ~
   ~:;bad ~]kitty." arg2) => "We have a bad kitty."

(setq arg 3)
(format nil "There ~[~1;was~2;is~3;will be~:;will have been~] ~
   a mess." arg) => "There will be a mess."
(format nil "There ~[~0,2:;was~3,5:;is~6,8;will be~
   ~:;will have been~] a mess." arg) => "There is a mess."

(setq *print-level* nil *print-length* 5)
(format nil "~:[ *PRINT-LEVEL* is defaulted~; *PRINT-LEVEL*=~D~]"
   *print-level*) => " *PRINT-LEVEL* is defaulted"
(format nil "~@[ *PRINT-LEVEL*=~D~]~@[ *PRINT-LENGTH*=~D~]"
   *print-level* *print-length*) => " *PRINT-LENGTH*=5"
```

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~S~
   ~:;~@{~#[~1; and~] ~S~^,~}~].")
(format nil foo) => "Items: none."
(format nil foo 'foo) => "Items: FOO."
(format nil foo 'foo 'bar) => "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz) => "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux) =>
"Items: FOO, BAR, BAZ, and QUUX."
```

~{*str***~}**

Provides an iteration mechanism. The corresponding argument of the **format** function should be a list, which is used as a set of arguments, as if for a recursive call to **format**. The terminator **~}** is only used in this construct.

The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes; if *str* uses up two arguments by itself, two elements of the list are used up each time around the loop.

If, before any iteration step, the list is empty, the iteration is terminated. Also, if a prefix parameter *n* is given, there are, at most, *n* repetitions of processing *str*.

If *str* is empty, an argument is used as *str*. It must be a string and must precede any arguments processed by the iteration. For example, the following are equivalent:

```
(apply #'format stream string args)
(format stream "~1{~:}" string args)
```

This uses **string** as a formatting string. The **~1{** says to process it at most once, and the **~:}** says to process it at least once. Therefore it is processed exactly once, using **args** as the arguments.

The format is:

~*n***:@{***str***~:}**

It expects its argument to be a list of items.

n	Performs, at most, *n* repetitions.
:	Performs iteration on a list of sublists. At each repetition step, one sublist is used as the set of arguments to *str*; on the next repetition a new sublist is used, whether or not all of the last sublist has been processed.
@	Performs iteration on the remaining elements in the arguments list. At each repetition step, the next element is used for an argument to *str*.
:@	Performs iteration on the remaining elements in the arguments list, each of which should be a list. At each repetition step, the next argument is used as a list of arguments to *str*.

~:} Forces *str* to be processed at least once even if the initial list of arguments is null (however, do not override an explicit prefix parameter of zero).

Examples:

```
(format nil "Here it is:~{ ~S~}." '(a b c)) =>
"Here it is: A B C."

(format nil "Pairs of things:~{ <~S,~S>~}."
   '(a 1 b 2 c 3)) =>
"Pairs of things: <A,1> <B,2> <C,3>."
(format nil "Pairs of things:~:{ <~S,~S>~}."
   '((a 1) (b 2) (c 3))) =>
"Pairs of things: <A,1> <B,2> <C,3>."
(format nil "Pairs of things:~@{ <~S,~S>~}."
   'a 1 'b 2 'c 3) =>
"Pairs of things: <A,1> <B,2> <C,3>."
(format nil "Pairs of things:~:@{ <~S,~S>~}."
   '(a 1) '(b 2) '(c 3)) =>
"Pairs of things: <A,1> <B,2> <C,3>."
```

As another example, the **format** function itself uses **format:format-error** to signal error messages, which in turn uses **zl:ferror**, which uses **format** recursively. **format:format-error** takes a string and arguments, like **format**, but also prints some additional information: if the control string in **ctl-string** actually is a string (it might be a list), it prints the string and a small arrow showing where, in the processing of the control string, the error occurred. The variable **ctl-index** points one character after the place of the error.

```
(defun format-error (string &rest args)
  (if (stringp ctl-string)
      (ferror nil "~1{~:}~%~VT↓~%~3X/""~A/""~%"
              string args (+ ctl-index 3) ctl-string)
      (ferror nil "~1{~:}" string args)))
```

This first processes the given string and arguments using **~1{~:}**, then tabs a variable amount for printing the down-arrow, then prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
         must be a number
                   ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
```

~<str~>

Justifies *str* within a field at least *mincol* wide. The next argument in *args* of the **format** function must be a string. The terminator **~>** is only used in this construct.

With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified. If there is only one segment, as a special case, it is right justified.

Note that *str* can include format directives. One of the examples illustrates how the ~**<** directive can be combined with the ~**f** directive to provide more advanced control over the formatting of numbers. Another example illustrates the use of ~^ within a ~**<** construct. ~^ eliminates the segment in which it appears and all following segments if there are no more arguments. If a segment contains a ~^ and **format** runs out of arguments, it stops there instead of getting an error, and that segment, as well as the rest of the segments, is ignored.

If the first clause of a ~**<** is terminated with ~**:;** instead of ~**;**, it is used in a special way. The first clause is ignored in performing the spacing and padding. When the padded result has been determined, if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, the text segment for the first clause is output before the padded text. The first clause should contain a carriage return (~%). The first clause is always processed, and so any arguments to which it refers are used; the decision is whether to use the resulting segment of text, not whether to process the first clause.

If the ~**:;** has a prefix parameter *n*, the padded text must fit on the current line with *n* character positions to spare, to avoid overprinting the first clause's text. For example, the following control string can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with "**;;** ".

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

The prefix parameter **1** in ~**1:;** accounts for the width of the comma that follows the justified item if it is not the last element in the list, or the period if it is. If ~**:;** has a second prefix parameter, it specifies the width of the line, overriding the natural line width of the output stream. To make the preceding example use a line width of 50, you would write:

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second parameter is not specified, **format** sees whether the stream handles the **:size-in-characters** message. If it does, **format** sends that message and uses the first returned value as the line width in characters. If it does not, **format** uses **72** as the line width.

Rather than using this complicated syntax, you can often use either **format:print-list**, or **format-textual-list**. See the function **format:print-list**. See the function **format-textual-list**.

The format is:

~mincol,colinc,minpad,padchar<str~;str~>

It expects its argument to be a list of items.

mincol — The total output width is at least *mincol*, filled out by *padchar*. The default is **0**.

colinc — Padding characters are inserted *colinc* characters at a time. The default is **1**.

minpad — The minimum number of copies of *padchar* to use between each segment. The default is **0**.

padchar — The character used for padding the field, if the field is wider than the object to be printed. This character should be specified by using an quote (') followed by the padding character. The default is the **#\space** character.

~; — Separates the strings into segments. Any spacing is evenly divided between the text segments.

~:; — The first clause is ignored in performing the spacing and padding. When the padded result has been determined, if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, the text segment for the first clause is output before the padded text. The first clause should contain a carriage return (*~%*). The first clause is always processed, and so any arguments to which it refers are used; the decision is whether to use the resulting segment of text, not whether to process the first clause.

: — Adds spacing before the text segments.

@ — Adds spacing after the last text segment.

:@ — Divides spacing evenly before the first segment, between each segment, and after the last segment.

Examples:

```
(format nil "~12<foo~;bar~>") => "foo      bar"
(format nil "~12:<foo~;bar~>") => "   foo   bar"
(format nil "~12:@<foo~;bar~>") => "  foo  bar  "
(format nil "~12<foobar~>") => "      foobar"
(format nil "~12:<foobar~>") => "       foobar"
(format nil "~12@<foobar~>") => "foobar      "
(format nil "~12:@<foobar~>") => "   foobar   "
(format nil "$~10,,,'*<~4f~>" 2.59023) => "$******2.59"

(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        => "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        => "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        => "FOO   BAR   BAZ"
```

```
(format nil "~%;; ~{~<~%;; ~1,50:;~S~>~^, ~}.~%"
   '(june july august september october november december)) =>
"
;; JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER,
;; NOVEMBER, DECEMBER.
"
```

~→str~←

Indents *str* at the cursor position that is current at the time of the ~→. The next argument in *args* of the **format** function must be a string. The terminator ~← is only used in this construct. ~→ and ~← can be nested like **~[~]** and **~<~>**; if they are nested, the indentation of an inner pair is relative to the margin set by the pair containing it. A numeric argument, if supplied, specifies how far to indent. This directive is especially useful in making error messages indent properly. This **format** directive is a Symbolics extension to Common Lisp.

The format is:

~n~→str~←

It expects its argument to be a string.

Examples:

```
(format t "~&Error:  ~→~A~←" "File not found
for FOO.LISP.1")
Error:  File not found
        for FOO.LISP.1
```

~(*str***~)**

Formats a string in appropriate case. With no prefixes, every uppercase character is mapped to the corresponding lowercase character. The **~(** directive must be matched by a corresponding **~)** directive.

The format is:

~:@(*str***~)**

It expects its argument to be a string.

:	Capitalizes all words (like **string-capitalize**).
@	Capitalizes the first word and force the rest to lowercase.
:@	Converts every lowercase character to the corresponding uppercase character.

Examples:

```
(format nil "~(~S~)" "fOo BaR") => "\"foo bar\""
(format nil "~:(~S~)" "fOo BaR") => "\"Foo Bar\""
(format nil "~@(~S~)" "fOo BaR") => "\"Foo bar\""
(format nil "~:@(~S~)" "fOo BaR") => "\"FOO BAR\""
```

~⊂str~⊃

Formats *str* in a specified character style. The character style can be specified as a prefix parameter to the directive, as in *~character-style⊂str~⊃*, or as an argument, using **V** (as explained below). See the section "Character Styles". The ~⊂ directive must be matched by a corresponding ~⊃ directive, which is only used in this construct. This **format** directive is a Symbolics extension to Common Lisp.

You can supply the character style parameter in the format control string as a single character, as in ~'i⊂...~⊃. In that case, the character should be one of the following:

'i	**:italic**
'b	**:bold**
'p	**:bold-italic**
'r	**:roman**

You can also have the character style parameter taken as an argument, using ~V⊂...~⊃. In that case, it may be a character style face code, like **:italic**; or else something acceptable to **si:parse-character-style**, such as a list like **(:fix :italic nil)** or an actual character style object. See the section "Character Styles".

The format is:

*~**:**⊂str~⊃*

It expects its argument to be a string.

: ~:⊂...~⊃ binds the line-height of the output stream. See the function **with-character-style**.

Examples:

```
(format nil "Moose bites can be ~'i⊂very~⊃ nasty, mind you.") =>
"Moose bites can be very nasty, mind you."
(format nil "Half the square root of ~'i⊂two~⊃ is ~v⊂~s~⊃."
        '(:fix :bold :normal) (sind 45)) =>
"Half the square root of two is 0.7071068."
```

~\\quoted-string

Prints a quoted string. The modifiers work as follows: A colon prints the quotes only if they are needed for the current readtable, and an at-sign uses a vertical bar (|) instead of a double-quote (") for quoting. For example:

```
(format t "~{~%~\\quoted-string\\ ~:\\quoted-string\\ ~
             ~@\\quoted-string\\ ~:@\\quoted-string\\~}~%"
          '("foo" "foo" "foo" "foo" "foo bar" "foo bar" "foo bar" "foo bar"))
"foo" foo |foo| foo
"foo bar" "foo bar" |foo bar| |foo bar|
```

~\\date

Prints its argument as a date and time, assuming the argument is a universal time. Writes the date out in words. It uses the function **time:print-universal-date**.

```
(format nil "Today is ~\\date\\" (time:get-universal-time)) =>
"Today is Friday the twenty-sixth of September, 1986; 5:56:19 pm"
```

~\\time

Prints its argument as a date and time, assuming the argument is a universal time. Writes the date in short form. It uses the function **time:print-universal-time**.

```
(format nil "Today is ~\\time\\" (time:get-universal-time)) =>
"Today is 9/26/86 17:58:05"
```

~\\datime

Prints the current date and time of day. It does not take an argument. It uses the function **time:print-current-time**.

```
(format nil "Today is ~\\datime\\") => "Today is 9/26/86 17:59:56"
```

~\\time-interval

Prints the length of a time interval. It uses the function **time:print-interval-or-never**.

```
(setq a (time:get-universal-time))
...
(format nil "It is ~\\time-interval\\ since I set this variable"
        (- (time:get-universal-time) a)) =>
"It is 3 days 7 minutes 18 seconds since I set this variable"
```

~\\presentation

Prints its argument via **present**. An object's presentation can define how it is printed, and this may be more readable than **(write object :escape nil)**. If so, **~\\presentation** is a better choice than **~A**. The presentation type can also affect the object's mouse sensitivity. See the section "Presentation Substrate Facilities".

The format is:

~@\\presentation

It expects its argument to be a Lisp object suitable for **present**.

@ Prints the object using the presentation type specified as the next element in the list of arguments.

Examples:

```
(format nil "~\\presentation\\" '(a b c)) => "(A B C)"
(format nil "~@\\presentation\\"
   '(a b c) '((sequence sys:expression))) => "A, B, and C"
```

Functions Related to format

format:defformat *directive (arg-type) arglist* &body *body*
Defines a new **format** directive.

format:print-list *destination element-format-string list* &optional *(separator-format-string* **",** *") (start-line-format-string* " *") (tilde-brace-options* "*")*
Provides a simpler interface for the specific purpose of printing comma-separated lists where no element from the list is broken at the end of a line.

sys:with-indentation *(stream-var relative-indentation)* &body *body*
Within the body of **sys:with-indentation**, any output to *stream-var* is preceded by a number of spaces. At every recursion, the additional indentation is specified by *relative-indentation*.

Streams

Introduction to Streams

Many programs accept input characters and produce output characters. Methods for performing input and output vary greatly from one device to another. Programs should be able to use any device available without each program having to know about each device.

The concept of *streams* solves this problem. A stream is a source and/or sink of data. A set of *operations* is available with every stream; operations include such actions as "output a character" and "input a character". The way to perform an operation to a stream is the same for all streams, although what happens inside a stream depends on the kind of stream it is. Thus a program needs to know only how to deal with streams in general.

In Genera, streams are implemented as flavors. You can operate on a stream by using generic functions or by sending it messages, depending on what type of oper-

ations the stream supports. Flavors, generic functions, and message-passing are described elsewhere: See the section "Flavors".

Some streams can do only input, some only output, and some can do both. Some streams support only some operations; however, unsupported operations might work, although slowly, because the **sys:stream-default-handler** can handle them. An operation called **:which-operations** returns a list of the names of all operations that are supported "natively" by a stream. (All streams support **:which-operations**, so it might not be in the list itself.)

Types of Streams

In addition to the streams documented in this chapter, Genera supports hardcopy streams, which are documented elsewhere.

See the section "Hardcopy Streams".

Standard Common Lisp Streams

Several variables whose values are streams are used by many functions in the Lisp system. By convention, variables that are expected to hold a stream capable of input have names ending with **-input**. Similarly, variables expected to hold a stream capable of output have names ending with **-output**. Those expected to hold a bidirectional stream have names ending with **-io**.

The variables ***standard-input***, ***standard-output***, ***error-output***, ***trace-output***, ***query-io***, and ***debug-io*** are initially bound to synonym streams that pass all operations on to the stream that is the value of ***terminal-io***. Thus any operation performed on those streams goes to the terminal.

No user program should ever change the value of ***terminal-io***. For example, a program to divert output to a file should do so by binding the value of ***standard-output***; that way, error messages sent to ***error-output*** can still get to the user by going through ***terminal-io***, which is usually what is desired.

standard-input	In the normal Lisp top-level loop, input is read from whatever stream is the value of ***standard-input***.
standard-output	In the normal Lisp top-level loop, output is sent to whatever stream is the value of ***standard-output***.
error-output	A stream to which error messages should be sent.
terminal-io	The stream that connects to the user's console.
query-io	A stream to be used when asking questions of the user.
debug-io	A stream to be used for interactive debugging purposes.
trace-output	The stream on which the **trace** function prints its output.

Standard Zetalisp Streams

The variables **zl:standard-input**, **zl:standard-output**, **zl:error-output**, **zl:trace-output**, and **zl:query-io** are initially bound to synonym streams that pass all operations on to the stream that is the value of **zl:terminal-io**. Thus any operation performed on those streams goes to the terminal.

These variables are synonyms for the Common Lisp variables with similar names. For example, **zl:standard-input** is a synonym for ***standard-input***. When writing new programs, you should use the Common Lisp variables instead of the Zetalisp variables.

zl:standard-input In your new programs, we recommend that you use the variable ***standard-input*** which is the Common Lisp equivalent of **zl:standard-input**.

zl:standard-output
In your new programs, we recommend that you use the variable ***standard-output*** which is the Common Lisp equivalent of **zl:standard-output**.

zl:error-output In your new programs, we recommend that you use the variable ***error-output*** which is the Common Lisp equivalent of **zl:error-output**.

zl:query-io In your new programs, we recommend that you use the variable ***query-io*** which is the Common Lisp equivalent of **zl:query-io**.

zl:terminal-io In your new programs, we recommend that you use the variable ***terminal-io*** which is the Common Lisp equivalent of **zl:terminal-io**.

zl:trace-output In your new programs, we recommend that you use the variable ***trace-output*** which is the Common Lisp equivalent of **zl:trace-output**.

zl:debug-io In your new programs, we recommend that you use the variable ***debug-io***, which is the Common Lisp equivalent of **zl:debug-io**.

dbg:*debug-io-override*
Diverts the Debugger to a stream that is known to work.

Coroutine Streams

Functions that produce data as output (output functions) are written in terms of **write-char** and other output operations. Functions that receive data as input (input functions) are written in terms of **read-char** and other input operations. Output functions operate on output streams, using the **write-char** function. Input functions operate on input streams, which use the **read-char** function. Sometimes it is desirable to view an output function as an input stream, or an input function as an output stream. You can do this with coroutine streams.

Here is a simplified explanation of how coroutine streams work. A coroutine input stream can be built from an output function. Whenever that stream sees the **read-char** function, it invokes the output function in a separate stack group so that the function can produce the data that **read-char** returns. A coroutine output stream can be built out of an input function; it works in the opposite fashion. Whenever the output stream sees **write-char**, it invokes the input function in a separate stack group so that the function can receive the data transmitted by **write-char**. It is also possible to connect functions that do both input and output, by using bidirectional coroutine streams. Since you can use coroutine streams to connect two functions, they are the logical inverse of **stream-copy-until-eof**, a function used to connect two streams.

To create a coroutine stream, use one of the following:

- If you want to make an input stream from an output function, use **sys:open-coroutine-stream** and give **:direction** the argument **:input**.
- If you want to make an output stream to an input function, use **sys:open-coroutine-stream** and give **:direction** the argument **:output**.
- If you want to make a bidirectional stream for a function that does both input and output, use **sys:open-coroutine-stream** and give **:direction** the argument **:io**.

Following is an example using a coroutine input stream:

```
(with-open-stream
  (input-stream
    (sys:open-coroutine-stream
      #'(lambda (output-stream)
          (with-open-stream (output-stream output-stream)
            (zl:print-disk-label 0 output-stream)))

      (read-line input-stream)
       =>  "1645 free, 260499//262144 used (99%)"
)
```

Following is an example using a coroutine output stream:

```
(with-open-stream
  (output-stream
    (sys:open-coroutine-stream
      #'(lambda (input-stream) (setq x (read input-stream))
                :direction :output))
    (write-string "(a b c)" output-stream))

  x => (A B C)
)
```

Coroutine streams are implemented as buffered character streams. Each function that makes a coroutine stream actually creates two streams and one new stack group. One stream is associated with the new stack group and the other stream with the stack group that is current when the stream-making function is called.

When you use **sys:open-coroutine-stream** to make input or output coroutine streams, one stream is an input stream and the other is an output stream; they share a common buffer. If you make bidirectional coroutine streams, both streams are bidirectional; the input buffer of each stream is the output buffer of the other.

When you use **sys:open-coroutine-stream** to make input streams, the output function runs in the new stack group. When you make output streams, the input function runs in the new stack group. With bidirectional streams, the function that does input or output runs in the new stack group.

In the case of creating input streams, for example, you typically use the **read-char** function on the input stream that **sys:open-coroutine-stream** returns. The output stream is associated with the new stack group. When the input stream sees **read-char**, the new stack group is resumed, and the output function runs in that stack group. The output function typically uses **write-char** on the output stream associated with the stack group from which **sys:open-coroutine-stream** was called. When the output stream sees **write-char**, the associated stack group is resumed. The data transmitted to the output stream become input to **read-char** via the buffer that the two streams share. Creating output streams and bidirectional streams work in an analogous fashion.

In addition to **read-char** and **write-char** coroutine streams support other standard input and output operations, such as **:line-in** and **:string-out**. Actually, the **:next-input-buffer** method of the input stream and the **:send-output-buffer** method of the output stream resume the new stack group, not the receipt of **read-char** and **write-char** functions. Because the streams are buffered, use **with-output-stream** to or close the stream when you're done. Or, you can send a **:force-output** message to an output stream to cause the new stack group to be resumed.

Do not confuse coroutine streams with pipes. Coroutine streams are used for intraprocess communication; pipes are used for interprocess communication.

Functions That Handle Coroutine Streams

To create a coroutine stream, use **sys:open-coroutine-stream**:

sys:open-coroutine-stream *function* &key *(:direction* **:input***) (:buffer-size* **1024***) (:element-type* **'character***) (:serving-stream-flavor* **'cli::coroutine-stream***)* &allow-other-keys
Creates either input streams, output streams, or bidirectional streams, each with a shared buffer, depending on the argument given to *:direction*.

Coroutine streams are implemented as instances of the following flavors:

si:coroutine-input-stream
Used to construct an input stream from a function written in terms of output operations.

si:coroutine-output-stream
Used to construct an output stream to a function written in terms of input operations.

si:coroutine-bidirectional-stream
Used to construct a bidirectional stream to a function written in terms of input and output operations.

When reading older code, you might see the following obsolete functions used:

sys:make-coroutine-input-stream *function* &rest *arguments*
This function is obsolete; use **sys:open-coroutine-stream** instead.

sys:make-coroutine-output-stream *function* &rest *arguments*
This function is obsolete. Use **sys:open-coroutine-stream** instead.

sys:make-coroutine-bidirectional-stream *function* &rest *arguments*
This function is obsolete. Use **sys:open-coroutine-stream** instead.

Direct Access File Streams

LMFS supports direct access file streams, which are designed to facilitate reading and writing data from many different points in a file. They are typically used to construct files organized into discrete extents or *records*, whose positions within a file are known by programs that access them in nonsequential order. Although this could be done with the **:set-pointer** message to input file streams, the direct access facility provides the following additional functions:

- Direct access to output files.
- Bidirectional file streams, which allow interspersed reading and writing of data to and from varied locations in a file.
- No use of network connections or file buffers during the time between data reading and the next call to position. In contrast, using the **:set-pointer** message with ordinary ("sequential") input file streams incurs a significant network and data transfer overhead if the program repeatedly positions, reads several bytes, and then computes for a time.

It is important to note that the default unit of transaction for the stream is any character, which is not supported by direct access file streams. The **:element-type** argument of the **open** operation should therefore be **'string-char**.

Direct Access Input File Streams

You can operate on direct access input file streams by sending the **:read-bytes** message:

:read-bytes Sent to a direct access input or bidirectional file stream, this requests the transfer of *n-bytes* bytes from position *file-position* of the file. The message itself does not return any data to the caller. It causes the stream to be positioned to that point in the file, and the transfer of *n-bytes* bytes to begin. An EOF is sent following the requested bytes. The bytes can then be read using **:tyi**, **:string-in**, or any of the standard input messages or functions.

Direct Access Output File Streams

You create direct access output to output and bidirectional direct access file streams by sending a **:set-pointer** message to the stream, and beginning to write bytes using standard messages, such as **:tyo**, **:string-out**, and so forth. The bytes are written to the file starting at the location requested, at successive file positions. Although you can extend the file in this manner, you cannot do a **:set-pointer** to *beyond* the current end of the file.

Direct access output, therefore, consists of sequences of **:set-pointer** messages and data output. Data are not guaranteed to actually appear in the file until either the stream is closed or a **:finish** message is sent to the stream. See the message **:finish**.

Direct Access Bidirectional File Streams

Bidirectional direct access file streams combine the features of direct access input and output file streams. Sequences of **:read-bytes** messages and reading data can be interspersed with sequences of **:set-pointer** messages and writing data. The stream is effectively switched between "input" and "output" states by the **:read-bytes** and **:set-pointer** messages. You cannot read data with **:tyi** or similar messages if a **:set-pointer** message has been sent to the stream since the last **:read-bytes** message. Similarly, you cannot write data with **:tyo** or similar messages unless a **:set-pointer** message has been sent to the stream since the last **:read-bytes** or **:tyi** messages, or similar operation.

When the EOF of a byte sequence requested with a **:read-bytes** message has been read for a bidirectional stream, the system frees network and buffering resources.

Effect of Character Set Translation on Direct Access File Streams

The Symbolics generic file access protocol was designed to provide access to ASCII-based file systems for Symbolics computers. Symbolics machines support 8-bit characters and have 256 characters in their character set. This results in difficulties when communicating with ASCII machines that have 7-bit characters.

The file server, on machines not using the Symbolics character set, is required to perform character translations for any character (not binary) opening. Some Symbolics characters expand to more than one ASCII character. Thus, for character files, when we speak of a given position in a file or the length of a file, we must specify whether we are speaking in *Symbolics units* or *server units*.

This causes major problems in file position reckoning. It is useless for the Symbolics machine (or other user side) to carefully monitor file position, counting characters, during output, when character translation is in effect. This is because the operating system interface for "position to point *x* in a file", which the server must use, operates in server units, but the Symbolics machine (or other user end) has counted in Symbolics units. The user end cannot try to second-guess the translation-counting process without losing host independence.

Since direct access file streams are designed for organized file position management, they are particularly susceptible to this problem. As with other file streams, it is only a problem when character files are used.

You can avoid this problem by always using binary files. If you must use character files, consider doing one of the following:

- Know the expansions of the Symbolics machine, that is, characters such as Return that do not expand into single host characters. Note that this sacrifices host independence.

- Do not use these characters. See the section "NFILE Character Set Translation". This section explains which characters are expanded on the Symbolics computer.

Compression Streams

Overview of the Compression System

The compression system is designed to make files smaller. It uses an adaptive compression algorithm based on Lempel-Ziv-Welch, or LZW coding, described in "A Technique for High-Performance Data Compression", by Terry A. Welch, *IEEE Computer*, June 1984, pp. 8-19. This documentation does not explain the details of that algorithm; you should consult the original paper if you want an in-depth understanding of how the compression works. This compression algorithm is compatible with UNIX compression. However, by default, files are written with a more descriptive preamble that includes information such as the element-type of the underlying stream that was compressed. For compatibility with UNIX hosts, Genera can read (automatically) and write (with specification of a keyword argument) UNIX-style compressed files.

Note: The Compression Substrate is preliminary in Genera 8.0. It might be radically altered in function and/or interface in future releases.

Some terminology: A *compressor* is some stream that compresses its input to decrease its redundancy. A *decompressor* is some stream that reverses this transformation to reconstruct the original data. Decompressed data is data that has been compressed and then decompressed; *uncompressed* data is data that has never been compressed at all. If the decompressor correctly decompressed some compressed data, then the decompressed data is exactly equal to the uncompressed data. The reason we are careful to keep the terms decompressed and uncompressed data sep-

arate is to make it easier to talk about issues such as performance (the time it takes to copy uncompressed vs. compressed data, for example). The other reason this terminology is important is so you look for the correct stream flavors: To invert the action of a compressing stream, look for its corresponding decompressing stream; there is no uncompressing stream.

The compression algorithm is a one-pass, stream-oriented compressor that can compress any arbitrary data without rewinding the stream. The input stream can be any multiple of eight bits wide; the output stream is always eight bits wide.

The amount of compression achieved depends on the redundancy of the data, hence on the entropy present. A purely random stream of bits will compress to a larger stream, since there is overhead in the compression protocol itself; however, most real data compresses fairly substantially. Text files typically compress to 40% of their original size; binary files (.bin or .ibin) typically compress to 60% of their original size; world loads typically compress about 50% of their original size.

Since the compression substrate is stream-oriented and obeys the stream protocol, a compressing or decompressing stream may be inserted between any two other streams in an application. However, since the compressed data is only useful to a decompressor, usually the compressing stream is the last stream in a chain of streams before the data gets written to permanent storage, and the decompressing stream is the first stream in a chain when the data is read back again.

The compressed data is representated as eight-bit binary bytes. If you store such data on a file server, the file server must understand that this is binary data and not apply any transformations to it that would change the data (such as performing character set translation or newline translation). Hence, if you write such data to, for example, a UNIX host and then use FTP on the UNIX host to move it somewhere else, you should be using binary mode. The Genera side handles the data correctly, of course, since binary streams are created; this is only a problem if you expect the compressed data to be of the same data type (for example, text) as the uncompressed data, and is exactly the same problem that any user of the UNIX `compress` program must be aware of. (In other words, the compressed data is like any other binary stream; Genera choses a binary stream to communicate with the file server, which likewise handles the stream as binary data. Reading the data back works the same way, though you must be careful that you are actually opening the stream as a eight-bit binary stream. If you manipulate the compressed data on the server, you must manipulate it as if it were any other binary data, not character data.)

Using the Compression Substrate

Before using the compression substrate, you must make two important decisions:

1. Whether you want Symbolics-style or UNIX-style compression.

2. Whether you want character set translation to be applied to the data before it is compressed or after it is uncompressed.

Choosing the Type of Data Compression

Currently, the actual stream of compressed data is essentially the same between the Genera and UNIX formats, with the exception of a preamble which differs in Symbolics-style and UNIX-style compression. However, Symbolics makes no guarantees that the data will always match this closely. In the future, different compression algorithms may be used that mean that compressed data is not even approximately the same across the two formats; however, this should be a transparent change, as no programs need to understand the exact format of Symbolics-style compression except the Genera-supplied decompressor.

The Symbolics-style preamble is useful because it is often essential to know what the underlying element-type of the uncompressed data was when opening a stream to write decompressed data from an input stream of compressed data. Also, it is useful to embed a version number in the compressed data so it is trivial to change algorithms to increase performance while still being to read old compressed data correctly. The Symbolics-style preamble embeds these parameters and many more into the preamble to allow correct reconstruction of the compressed data.

The UNIX-style preamble, on the other hand, is a fixed-size (three bytes) preamble, of which the first two bytes are simply a magic value to identify this as a compressed file, followed by a byte whose sixth bit indicates whether so-called "block compression" is in effect, and whose low five bits indicate the number of bits used in keying the internal hash table. This information is completely insufficient if an application needs to know what sort of data was originally stored in its now-compressed form. UNIX copes with this by treating everything as eight-bit bytes; Genera's more flexible generic stream and network requires more sophisticated information.

Choosing When Character Set Translation is Applied to Data

To make it easier to interoperate with UNIX hosts, the compression substrate will also do character set translation between the Symbolics character set and ASCII if the **:unix-translation-in-effect?** keyword is non-**nil** when the compressing or decompressing stream is instantiated. You should set this keyword non-**nil** if you are compressing or decompressing data which is a subtype of **'character**. Do not specify translation if the data is binary (e.g., a UNIX tar file or some similar object).

compression::*likely-unix-binary-formats* *Variable*

A list of the file types that Compress File and Decompress File assume are binary files and do not need character set translation. This is only important if you are reading or writing a file with a UNIX-style compression preamble and are using :Translation Strategy Heuristicate. Users are encouraged to add or remove items from this list.

The current value of this variable is:

```
"TAR" "ARC" "ZOO" "LZH" "ZIP" "MID"
"GIF" "IFF" "TIF" "TIFF" "GEM" "NEO"
"SPC" "LIB" "OLB" "GL"
```

Compress File Command

Compress File *input-files output-files keywords*

Compresses the data in *input-files* and produces *output-files*. Wildcards are allowed. If *input-files* and *output-files* are the same files, the input files are replaced by the output files.

input-files {*pathname(s)*} One or more files to compress.

output-files {*pathname(s)*} One or more files to contain the compressed data.

keywords :Copy Properties, :Create Directories, :More Processing, :Output Destination :Preamble Type, :Query, :Translation Strategy

:Copy Properties {*list of file properties*} The properties you want duplicated in the new files. The default is author and creation date.

:Create Directories
{Yes, Error, Query} What to do if the destination directory does not exist. The default is Query.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window} Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Preamble Type {Symbolics, UNIX} Type of preamble to use.

:Query {Yes, No, Ask} Whether to ask before compressing each file.

:Translation Strategy
{Text, Binary, Query, Heuristicate} Whether or not to perform character set translation. Text means to do ASCII character set translation, reading each input file as a text file. Binary means not to do ASCII character set translation, reading each input

file as a binary file. Query asks, for each file, whether to treat the file as text or binary. Heuristicate attempts to guess whether the file is text or binary based on its name, as follows: The filename is broken up into *words*, where each word is separated by a non-alphanumeric character. A rightmost word of "Z" is removed (if present). Then the current rightmost word is checked against **compression::*likely-unix-binary-formats*** If a match is found, the file is assumed to be binary, otherwise it is assumed to be text.

:Translation Strategy is only useful if you are reading or writing a file with a UNIX-style compression preamble, because Symbolics-style compression preambles record the element type and character set of the compressed data. Using :Translation Strategy with a file having a Symbolics-style compression preamble is ignored with a warning.

Decompress File Command

Decompress File *input-files output-files keywords*

Decompresses the data in *input-files* and produces *output-files*. Wildcards are allowed. If *input-files* and *output-files* are the same files, the input files are replaced by the output files.

input-files {*pathname(s)*} One or more files to decompress.

output-files {*pathname(s)*} One or more files to contain the decompressed data.

keywords :Copy Properties, :Create Directories, :More Processing, :Output Destination :Preamble Type, :Query, :Translation Strategy

:Copy Properties {*list of file properties*} The properties you want duplicated in the new files. The default is author and creation-date.

:Create Directories
{Yes, Error, Query} What to do if the destination directory does not exist. The default is Query.

:More Processing {Default, Yes, No} Controls whether **More** processing at end of page is enabled during output to interactive streams. The default is Default. If No, output from this command is not subject to **More** processing. If Default, output from this command is subject to the prevailing setting of **More** processing for the window. If Yes, output from this command is subject to **More** processing unless it was disabled globally (see the section "FUNCTION M").

:Output Destination
{Buffer, File, Kill Ring, None, Printer, Stream, Window}

Where to redirect the typeout done by this command. The default is the stream ***standard-output***.

:Query {Yes, No, Ask} Whether to ask before decompressing each file.

:Translation Strategy
{Text, Binary, Query, Heuristicate} Whether or not to perform character set translation. Text means to do ASCII character set translation, writing each resulting file as a text file. Binary means not to do ASCII character set translation, writing each resulting file as a binary file. Query asks, for each file, whether to treat the file as text or binary. Heuristicate attempts to guess whether the file is text or binary based on its name, as follows: The filename is broken up into *words*, where each word is separated by a non-alphanumeric character. A rightmost word of "Z" is removed (if present). Then the current rightmost word is checked against **compression::*likely-unix-binary-formats***, and if a match is found, the file is assumed to be binary, else it is assumed to be text.

:Translation Strategy is only useful if you are reading or writing a file with a UNIX-style compression preamble, because Symbolics-style compression preambles record the element type and character set of the compressed data. Using :Translation Strategy with a file having a Symbolics-style compression preamble is ignored with a warning.

Compressing Data - Details and Examples

The compressor and decompressor are streams. You can use them by creating a stream instance of an appropriate flavor and passing input to or receive output from the instance. The compressor and decompressor perform in accordance with the init keywords you specify in the **make-instance** form.

Symbolics recommends that you do not bind the variety of global variables influencing the defaults for the various keywords. Instead, you can specify non-default values by specifying keywords in the call to **make-instance**, not by binding these global variables.

Use caution in choosing keyword values when constructing the compressing stream. The decompressing stream usually sets up most of its parameters using the Symbolics preamble and therefore does not need you to specify much in the way of keywords. Note that the compressor enables you to specify many meaningless combinations but malfunctions when you attempt to use them.

Stream Flavors for Compression and Decompression

Two stream flavors are available in compressing and decompressing data:

- **compression:buffered-compressing-output-stream**
- **compression:buffered-decompressing-input-stream**

Both streams are buffered and obey the protocol for buffered streams. Note that buffered streams conduct operations one buffer of data at a time, requiring you to specify the type of data a buffer can store. When using compression or decompression streams, specify the type of buffer the streams create in accepting input data or producing output data based on the stream receiving the input or producing the output.

Compression Example

Consider the following example and note that you have to compile the example in the **compression** package.

```
(defun COMPRESS (from to)
  (with-open-file (outstream to :direction :output :element-type '(unsigned-byte 8))
    (with-open-stream (the-stream (make-instance 'buffered-compressing-output-stream
                                        :stream outstream
                                        :uncompressed-stream-element-type 'string-char
                                        :public-buffer-element-type 'string-char))
      (with-open-file (instream from :direction :input :element-type 'string-char)
        (stream-copy-until-eof instream the-stream))))
  (values))
```

compress creates the output file with an element-type of **'(unsigned-byte 8)** (compressed data is always treated as eight-bit binary data). **compress** then creates the compressing stream. The **:stream** argument enables you to hook the compressing stream's output to the output stream written to stable storage. These keywords are *required* in any compressing stream:

:uncompressed-stream-element-type
: Describes the data before compression, and is dependent on the file type opened.

:public-buffer-element-type
: Specifies the type of buffer the compressing stream makes available for the input stream to fill. In other words, we have an input stream (in this case, **instream**) which returns data of a particular type (for example, **string-char** or **character** or **'(unsigned-byte 16)**). The **stream-copy-until-eof** identifies the stream as a buffered stream and sends it a **:new-output-buffer** message, and attempts to copy from the input stream into the buffer provided by the **:new-output-buffer**. Note that the data type in **instream** must match the type for each element of the buffer provided by **the-stream**'s response to a **:new-output-**

buffer message.

compress creates the input stream by opening the input file and copying data from the input stream into the compressing stream. The compressing stream outputs data to the final output stream which is written out to the file server.

You may wonder why both **:uncompressed-stream-element-type** and **:public-buffer-element-type** are needed. Consider an application where we are reading styled characters, whose type is **character**. In order to notify the decompressor that the data it is receiving consists of styled characters, you specify **:uncompressed-stream-element-type 'character**. However, you cannot read data of type **'character** from a file system. True styled characters are 32-bit wide quantities whose style bits are not constant from one world load to another. (For more information, see the section "Fields of a Character".) Therefore, you must read and write styled characters to permanent storage by writing them as objects of type **'string-char**. You have to write the styled characters even though the streams convert styled characters to sequences of epsilons and unstyled characters (these translating streams are automatically used when writing styled characters as **'string-char** characters and are normally invisible). Note that when compressing a file containing styled characters (which actually contains unstyled characters and some epsilon characters), you can specify **:public-buffer-element-type** and **:uncompressed-stream-element-type** as both **'string-char**. This is because you would simply read and write the same unstyled characters that the file system supplies, and the result is the same as if they were first read as epsilons-and-unstyled-characters, converted to true styled (fat) characters, converted back to epsilons-and-unstyled-characters, and written back out again.

Consider a case that arises in the Distribution dumper and reloader (the Distribute Systems and Restore Distribution activities), which deal with streams from some file server, compressing streams, and tape streams simultaneously. For example, the distribution dumper always sends eight-bit binary data to the tape stream, so **:public-buffer-element-type** must always be **'(unsigned-byte 8)**. (In other words, the tape stream always expects us to give it buffers in which each element type is **'(unsigned-byte 8)**.) However, files as read from the file system may have any element type. Therefore, the Distribution dumper creates the compressing stream with a value for **:uncompressed-stream-element-type** obtained by calling **stream-element-type** on the file system stream and using whatever that returns.

Consider this example where a function compresses .bin and .ibin files, knowing that such files are always of element-type **'(unsigned-byte 16)**:

```
(defun COMPRESS-BINARY-NATIVE-WIDTH (from to)
  (with-open-file (outstream to :direction :output :element-type '(unsigned-byte 8))
    (with-open-stream (compression-stream (make-instance 'buffered-compressing-output-
                                                         stream
                                                          :stream outstream
                                          :uncompressed-stream-element-type '(unsigned-byte 16)
                                          :public-buffer-element-type '(unsigned-byte 16)))
      (with-open-file (instream from :direction :input :element-type '(unsigned-byte 16))
        (fs:stream-copy-until-eof instream compression-stream))))
  (values))
```

Working backwards from the element-type of the input file, the input is 16-bit binary data requiring **instream** to be of that type. **:public-buffer-element-type** must also be of that type to enable copying from stream to stream. Since the eventual decompressor of this data has to know how wide the original stream was, **:uncompressed-stream-element-type** must also be of this type. Finally, note that **outstream** is still opened with element-type **'(unsigned-byte 8)**, since compressed data is *always* an 8-bit binary stream.

Decompression Example

```
(defun DECOMPRESS (input output)
  (with-open-file (instream input
                            :direction :input
                            :element-type '(unsigned-byte 8))
    (with-open-stream (the-stream
                       (make-instance 'buffered-decompressing-input-stream
                                      :stream instream))
      (with-open-file (outstream output
                                 :direction :output
                                 :element-type (cl:stream-element-type the-stream))
        (stream-copy-until-eof the-stream outstream))))
  (values))
```

This example opens the compressed file, creates a decompressing stream that reads from the stream of compressed data, creates an output file stream that will write the decompressed data, and copies. Note that the element-type of the output stream is derived from the element-type of the original (uncompressed) data, by asking the decompressing stream itself via **stream-element-type**.

Note that **:uncompressed-stream-element-type** is *not supplied* in creating the decompressing stream. The decompressor has no way of forcing the element-type of the original, uncompressed data. Instead, the decompressing stream learns that element-type by examining information in the preamble (if it's a Symbolics-style preamble), or by assuming that the original element-type of the uncompressed data was **'string-char** (if it's a UNIX-style preamble). This example assumes that the output file will always be of type **'string-char**. (If you are decompressing UNIX data which is really a binary file, you may use **:public-buffer-element-type '(unsigned-byte 8)** to ensure that you actually get a binary stream back from the decompressor.)

If you do not supply **:public-buffer-element-type** when creating a decompressing stream, the decompressor attempts to heuristicate an appropriate element-type by parsing the preamble. You may then determine what sort of buffer will be used by evaluating (hence the element-type of each individual item returned by the decompressing stream) by simply sending the stream an **:element-type** message, or by using the generic function **stream-element-type**. This is usually the recommended approach.

If you have an application which requires a particular type of element-type returned from the decompressing stream regardless of what the element-type of the uncompressed data was, you can specify a value for **:public-buffer-element-type** when you create the decompressing stream. This is often useful when the incoming data is known to be binary, but may be, for example, either eight or sixteen bits wide, whereas your application (a tape stream, for example) may only be able to cope with eight-bit data. Rather than using **fs:stream-copy-16-to-8** or similar functions, you can ask the decompressing stream to do this for you.

Note that, if you specify **:public-buffer-element-type**, the value returned by sending an **:element-type** message to the decompressing stream does not change. This means that you will have created a stream which could return data which is not the same element-type as returned by the **:element-type** message. This is not recommended, and is one reason why **:public-buffer-element-type** should only be used in unusual circumstances.

The act of creating the decompressing stream (that is, evaluating the **make-instance** form) causes the preamble to be read from the compressed data. The decompressor now knows which **:public-buffer-element-type** and **:uncompressed-stream-element-type** are appropriate before it reads any real data (e.g., after the preamble) from the stream. The stream supplied to the **:stream** keyword must be available to be read when the decompressing stream is instantiated.

Interoperating with UNIX Compression

To specify that the compressor should translate lisp machine characters into ASCII, including UNIX-style newline translation, include **:unix-translation-in-effect? t** when you create the compressing stream. The uncompressed data must be some subtype of **'character**. If reading data compressed with a Symbolics-style preamble, the decompressor determines from reading the preamble whether character set translation was performed and uses this information in deciding whether to translate back. If reading data compressed with a UNIX-style preamble, you must specify whether you want character set translation when decompressing the data. In general, this is appropriate for UNIX files which contain characters, and not appropriate for binary files such as tar files

In most cases, translating from the lisp machine character set to ASCII is only half the story. Having compressed ASCII data is important when interoperating with UNIX, which cannot understand Symbolics-preamble compressed data. Hence, if you expect UNIX to run `uncompress` or `zcat` on your compressed data, you must ensure that a UNIX-style preamble is written. You can specify a UNIX-style preamble by specifying **:preamble-type :unix** (the default is **:symbolics**) when creating the compressing stream.

It is not recommended that you use UNIX-style compression by default in code you write, since UNIX-style compression loses a great deal of information (such as the element-type of the original data) that is easier to retain rather than to reconstruct after the fact. However, if you write data that UNIX utilities must be able to decompress, using UNIX-style preambles is mandatory, and should work adequately, since UNIX utilities simply assume that they are getting data which is already of the right type.

Using Compression/Decompression with Unrewindable or Uncloseable Streams

You can compress or decompress data embedded in a larger stream. For example, the Distribute Systems activity writes compressed files to a distribution tape. The actual compressed data is written to a tape stream which is never closed until the tape is completely written. When Restore Distribution reads the compressed data, it cannot close or rewind the stream until the tape is completely read. However, each file compresses individually, enabling you to read particular files from the tape without having to read the entire tape.

These restrictions pose additional complications for the compression substrate. The compression substrate depends on knowing when all data of a particular stream being compressed is read. When writing compressed data to tape, you cannot close the compressing stream at the end of a file since closing that stream (which feeds into the tape stream) prematurely closes the tape stream.

Consider this example: assume you are compressing 8-bit binary data. Even though the uncompressed data is eight bits wide, the compressor does not preserve any particular byte-for-byte correspondence between compressed output and uncompressed input. The compressor transforms the incoming bytes into a stream of objects of varying sizes, usually ranging from 9 to 16 bits wide. It then packs these variable-sized objects into an 8-bit-wide output stream without wasting any bits in padding. The compressor has a few "leftover bits" that could not go into the last byte produced (because it was already full), but cannot be put in a byte of its own because there is more data waiting to be compressed contributing additional bits of output. The compressor writes output data bytes asynchronously from input bytes. Under normal circumstances, the compressor only outputs "leftover bits" in its own byte if there is no additional input. The compressor cannot determine whether additional input is present until it encounters the end of the input stream.

Problems to Consider

It is inconvenient if the compressor simply outputs its final byte of output when encountering the end of the input stream. It may occasionally be useful to copy more than one file into the same compressed output, for example, which would require complicated mechanisms (probably ruling out use of **stream-copy-until-eof**) to ensure that the end of the compressor's input stream occurred only once, at the very end. These problems are also present when decompressing, in which the decompressor may be decompressing from a tape stream containing several files, all separately compressed. The decompressor does not know when it is done until it encounters the end of input stream. The tape stream never returns an end-of-stream indication until we reach the end of the tape.

Solutions

You can avoid these problems using **compression::finalize-output** and **compression::finalize-input**. These routines accomplish the work involved in writing or reading these leftover bits. The **:close** methods for the compressing and decompressing streams call these functions (which do not work if they've already been called). However, if you have an application that cannot close the compressing or decompressing stream when you have processed all the relevant data, you can call the appropriate routine and flush the pending state in the compression substrate to the output stream. For example:

```
(let ((compression-stream
            (make-instance 'compression::buffered-compressing-output-stream
                :stream tape-stream
                :uncompressed-stream-element-type data-element-type
                :public-buffer-element-type `(cl:unsigned-byte data-byte-size))))
      (stream-copy-until-eof data compression-stream)
      (compression::finalize-output compression-stream nil))
```

(The second argument to **compression::finalize-output**, **nil**, determines whether the output is closed in **:abort** mode. The **:close** method for the stream uses this information, setting this value to non-**nil** if the stream is being aborted).

Stream Operations

Making Your Own Stream

Functions for Creating Streams

This section summarizes some standard functions that make streams for you, such as **make-synonym-stream** and **make-two-way-stream**. For examples of using these functions to make a stream, see the section "Examples of Making Your Own Stream".

Genera also offers forms that allow you to evaluate Lisp forms while performing input or output on a stream, such as **with-open-stream** and **zl-user:with-input-from-stream**. For more information on **with-open-...** forms: See the section "Accessing Files".

make-concatenated-stream &rest *streams*
: Returns a stream that only works in the input direction.

make-echo-stream *input-stream output-stream*
: Not currently available.

make-string-input-stream *string* &optional *(start* **0***) end*
: Returns an input stream.

make-string-output-stream
: Returns an output stream that will accumulate all string output given it.

get-output-stream-string *stream*
Returns a string containing all of the characters output to *stream* so far.

make-synonym-stream *stream-symbol*
Creates and returns a "synonym stream".

zl:make-syn-stream *stream-symbol*
zl:make-syn-stream creates and returns a "synonym stream" (syn for short).

make-two-way-stream *input-stream output-stream*
Returns a bidirectional stream that gets its input from *input-stream* and sends its output to *output-stream*.

sys:stream-default-handler *fctn op arg1 args*
Tries to handle the *op* operation on *stream*.

sys:null-stream *op* &rest *args*
Used as a dummy stream object.

Examples of Making Your Own Stream

You can also write your own streams. Here is a sample output stream that accepts characters and conses them onto a list.

```
(defvar *the-list* nil)
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 *the-list*)))
        (:which-operations '(:tyo))
        (otherwise
          (stream-default-handler (function list-output-stream)
                                  op arg1 rest))))
```

The lambda-list for a stream must always have one required parameter (**op**), one optional parameter (**arg1**), and a rest parameter (**rest**). This allows an arbitrary numbe<r of arguments to be passed to the default handler. This is an output stream, so it supports the **:tyo** operation. Note that all streams must support **:which-operations**. If the operation is not one that the stream understands (for example, **:string-out**), it calls the **sys:stream-default-handler**. The calling of the default handler is required, since the willingness to accept **:tyo** indicates to the caller that **:string-out** will work.

Here is a typical input stream that generates successive characters of a list.

```
(defvar *the-list*)     ;Put your input list here
(defvar untyied-char nil)
(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi
     (cond ((not (null untyied-char))
            (prog1 untyied-char (setq untyied-char nil)))
           ((null *the-list*)
            (and arg1 (error arg1)))
           (t (prog1 (car *the-list*)
                     (setq *the-list* (cdr *the-list*))))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler (function list-input-stream)
                              op arg1 rest))))
```

The important things to note are that **:untyi** must be supported, and that the stream must check for having reached the end of the information and do the right thing with the argument to the **:tyi** operation.

The above stream uses a free variable (***the-list***) to hold the list of characters, and another one (**untyied-char**) to hold the **:untyi**ed character (if any). You might want to have several instances of this type of stream, without their interfering with one another. This is a typical example of the usefulness of closures in defining streams. The following function will take a list and return a stream that generates successive characters of that list.

```
(defun make-a-list-input-stream (list)
   (let-closed ((*the-list* list) (untyied-char nil))
       (function list-input-stream)))
```

The above streams are very simple. When designing a more complex stream, it is useful to have some tools to aid in the task. The **defselect** function aids in defining message-receiving functions. The Flavor System provides powerful and elaborate facilities for programming message-receiving objects. See the section "Flavors".

General Stream Functions

streamp *x*
: Returns **t** if *x* is a stream, and otherwise it returns **nil**.

input-stream-p *stream*
: Returns *t* if *stream* can handle input operations, otherwise returns **nil**.

output-stream-p *stream*
: Returns **t** if *stream* can handle output operations, and otherwise it returns **nil**.

stream-element-type *stream*
Returns a type specifier which indicates what objects can be read from or written to *stream*.

General-Purpose Stream Operations

Basic General-Purpose Stream Operations

:tyo *char* Puts the *char* into the stream.

:tyi Gets the next character from the stream and returns it.

:untyi The stream will remember the character *char*, and the next time a character is input, it will return the saved character.

:which-operations The object should return a list of the messages and generic functions it can handle.

:operation-handled-p
The object should return **t** if it has a handler for the specified operation, **nil** if it does not.

:send-if-handles The object should perform the operation (whether generic function or message) if it has a method for it.

:characters Returns **t** if the stream is a character stream, **nil** if it is a binary stream.

:direction Returns one of the keyword symbols **:input**, **:output**, or **:bidirectional**.

:interactive The **:interactive** message to a stream returns **t** if the stream is interactive and **nil** if it is not.

Advanced General-Purpose Stream Operations

Any stream must either support **:tyo** or support both **:tyi** and **:untyi**. Several more advanced input and output operations work on any stream that can do input or output (respectively). Some streams support these operations themselves; you can tell by looking at the list returned by the **:which-operations** operation. Others are handled by the "stream default handler" even if the stream does not know about the operation itself. However, in order for the default handler to do one of the more advanced output operations, the stream must support **:tyo**, and for the input operations the stream must support **:tyi** (and **:untyi**).

Here is the list of such operations:

:input-wait Message to an input stream causes the stream to **process-wait** with *whostate*.

:listen The main purpose of **:listen** is to test whether the user has pressed a key.

:tyipeek On an input stream, returns the next character that is about to be read, or **nil** if the stream is at end-of-file.

:fresh-line Tells the stream to position itself at the beginning of a new line.

:clear-rest-of-line Erases from the current position to the end of the current line.

:string-out The characters of *string* are successively output to the stream.

:line-out The characters of *string*, followed by a carriage return character, are output to the stream.

:string-in Reads characters from an input stream into *vector*, using the sub-vector delimited by *start* and *end*.

:line-in The stream should input one line from the input source and return it as a string with the carriage return character stripped off.

:string-line-in Allows you to read many lines successively into the same buffer without creating strings.

:clear-input The stream clears any buffered input.

:clear-output The stream clears any buffered output.

:force-output Causes any buffered output to be sent to a buffered asynchronous device.

:finish Does a **:force-output** to a buffered asynchronous device.

:close Closes a stream.

:eof Indicates the end of data on an output stream.

Special-Purpose Stream Operations

Basic Special-Purpose Stream Operations

See the section "General-Purpose Stream Operations". There are several other defined operations that the default handler cannot deal with; if the stream does not support the operation itself, sending that message causes an error. This section describes the most commonly used, least device-dependent stream operations. Windows, files, and Chaosnet connections have their own special stream operations, which are documented separately.

:input-editor Supported by interactive streams such as windows.

:beep Attracts the attention of the user by making an audible beep and/or flashing the screen.

:tyi-no-hang Identical to **:tyi** except that if it would be necessary to wait in order to get the character, returns **nil** instead.

:untyo-mark This is used by the grinder if the output stream supports it.

:untyo This is used by the grinder in conjunction with **:untyo-mark**.

Special-Purpose Operations for Window Streams

The following operations are implemented by window streams only. There are many other special-purpose stream operations for graphics. See the section "Using the Window System".

:read-cursorpos Returns two values, the current *x* and *y* coordinates of the cursor.

:set-cursorpos Sets the position of the cursor.

:increment-cursorpos
Sets the position of the cursor; *x* and *y* are the amounts to increment the current *x* and *y* coordinates.

:home-cursorpos Sets the position of the cursor and puts the cursor back at the begining of the stream.

:clear-window Erases the window on which this stream displays.

:clear-rest-of-window
Erases from the current position to the end of the current window.

Special-Purpose Operations for Buffered Input Streams

The following operations are implemented by buffered input streams. They allow increased efficiency by making the stream's internal buffer available to the user.

:read-input-buffer Returns three values: a buffer array, the index in that array of the next input byte, and the index in that array just past the last available input byte.

:advance-input-buffer
If *new-pointer* is non-**nil**, it is the index in the buffer array of the next byte to be read. If *new-pointer* is **nil**, the entire buffer has been used up.

Special-Purpose Operations for Buffered Output Streams

The following operations are provided for buffered output streams. They allow you to hand the stream's output buffer to a function that can fill it up.

:get-output-buffer Returns an array and starting and ending indices.

:advance-output-buffer
Says that the array returned by the last **:get-output-buffer** operation was filled up through *index*.

File Stream Operations

The following messages can be sent to file streams, in addition to the normal I/O messages that work on all streams. Note that several of these messages are useful to send to a file stream which has been closed. Also note that some of these messages do not work for some file systems, because the host file system determines which messages are valid. See the section "Naming of Files".

:pathname Returning the pathname opened to get a stream.

:truename Returning the pathname of a file opened on a stream.

:length Return the length of a file in bytes or characters.

:characters Returns **t** if the stream is a character stream, **nil** if it is a binary stream.

:creation-date Returns the creation date of the file, as a number which is a universal time.

:info Returns a cons of the truename and creation date of the file.

:delete Deletes an open file in a stream.

:rename Renames an open file in a stream.

:properties A list whose car is the pathname of a file and whose cdr is a list of the properties of the same file.

:change-properties
Changing the file properties of a file open on a stream.

File output streams implement the **:finish** and **:force-output** messages.

:finish Does a **:force-output** to a buffered asynchronous device.

:force-output Causes any buffered output to be sent to a buffered asynchronous device.

The following operations are implemented by streams to random-access devices, principally files.

:read-pointer Returns the current position within the file, in characters (bytes in fixnum mode).

:set-pointer Sets the reading position within the file to *new-pointer* (bytes in fixnum mode).

Network Stream Operations

:connected-p Returns **t** if the stream is fully connected to an active network connection, **nil** otherwise.

:start-open-auxiliary-stream
This message is sent to a stream to establish another stream,

via another connection, over the same network medium, to the same host. It is used for either end of the connection.

:complete-connection
This message is sent to a new stream created by **:start-open-auxiliary-stream**, in order to wait for the connection to be fully established. **:complete-connection** is used whether or not this side is active.

:set-input-interrupt-function
This message assigns a *function* to be applied to any *args* whenever input becomes available on the connection, or the connection goes into an unusable state.

The :read and :print Stream Operations

A stream can specially handle the reading and printing of objects by handling the **:read** and **:print** stream operations. Note that these operations are optional and that most streams do not support them.

If the **zl:read** function is given a stream that has **:read** in its which-operations, then instead of reading in the normal way it sends the **:read** message to the stream with one argument, **zl:read**'s *eof-option* if it had one or a magic internal marker if it did not. Whatever the stream returns is what **zl:read** returns. If the stream wants to implement the **:read** operation by internally calling **zl:read**, it must use a different stream that does not have **:read** in its which-operations.

If a stream has **:print** in its which-operations, it can intercept all object printing operations, including those due to the **print**, **prin1**, and **princ** functions, those due to **format**, and those used internally, for instance in printing the elements of a list. The stream receives the **:print** message with three arguments: the object being printed, the *prindepth* (for comparison against the **zl:prinlevel** variable), and *slashify-p* (**t** for **prin1**, **nil** for **princ**). If the stream returns **nil**, then normal printing takes place as usual. If the stream returns non-**nil**, then **print** does nothing; the stream is assumed to have output an appropriate printed representation for the object. The two following functions are useful in this connection; however, they are in the **system-internals** package and might be changed without much notice.