LMI

LAMBDA

SYSTEM MAP for Release 2.0
** indicates location of tab divider in binder

These manuals are part of your Lambda documentation, but are not part of a binder.

Intro to Lambda
ZetaLISP-Plus Commands

Here are the binders and their contents:

I

LISP Machine Manual, Part 2

Sixth Edition, System Version 99

June 1984

Richard Stallman
Daniel Weinreb
David Moon

Distributed by LMI 6033 W. Century Blvd. Los Angeles CA 90045 USA

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advance Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-80-C-0505.

21. Objects, Message Passing, and Flavors

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in Zetalisp and used by the Lisp Machine software system. Its purpose is to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure-calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter attempts to explain what programming with objects and with message passing means, the various means of implementing these in Zetalisp, and when you should use them. It assumes no prior knowledge of any other languages.

21.1 Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*, conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be "pieces of text", "pointers into text", and "display windows". In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows". After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on "pieces of text" might include inserting text and deleting text; operations on "pointers into text" might include moving forward and backward; and operations on "display windows" might include redisplaying the window and changing which "piece of text" the window is associated with.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of the type.

The new types may exist only in the programmer's mind. For example, it is possible to think of a disembodied property list as an abstract data type on which certain operations such as **get** and **putprop** are defined. This type can be instantiated with **(cons nil nil)** (that is, by evaluating this form you can create a new disembodied property list); the operations are invoked through functions defined just for that purpose. The fact that disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view. However, such conceptual data types cannot be distinguished automatically by the system; one cannot ask "is this object a disembodied property list, as opposed to an ordinary list".

The **defstruct** for **ship** early in chapter 20 defines another conceptual type. **defstruct** automatically defines some operations on this object, the operations to access its elements. We could define other functions that did useful things with **ship**'s, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one Lisp object. The Lisp object we use for the representation has *structure* and refers to other Lisp objects. In the disembodied property list case, the Lisp object is a list of pairs; in the **ship** case, the Lisp object is an array whose details are taken care of by **defstruct**. In both cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. **get** examines the state of a property list, and **putprop** alters it; **ship-x-position** examines the state of a ship, and **(setf (ship-x-position** *ship***) 5.0)** alters it.

We have now seen the essence of object-oriented programming. A conceptual object is modeled by a single Lisp object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.

21.2 Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques available, they help and encourage you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the program knows what the facility's external interfaces guarantee to do, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program only depends on the external definition of these operations: it knows that if it **putprop**'s a property, and doesn't **remprop** it (or **putprop** over it), then it can do **get** and be sure of getting back the same thing it put in. The important thing about this hiding of the details of the implementation is that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand what they undertake to do. This saves the programmer a lot of time and lets him concentrate his energies on understanding the program he is working on. Another good thing about this hiding is that the representation of property lists could be changed and the program would continue to work. For example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the **ship** example. The caller is presented with a collection of operations, such as **ship-x-position**, **ship-y-position**, **ship-speed**, and **ship-direction**; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, **ship-x-position** and **ship-y-position** would be accessor functions, defined automatically by **defstruct**, while **ship-speed** and **ship-direction** would be functions defined by the implementor of the ship type. The code might look like this:

```
(defstruct (ship :conc-name)
  x-position
  y-position
  x-velocity
  y-velocity
  mass)

(defun ship-speed (ship)
  (sqrt (+ (^ (ship-x-velocity ship) 2)
           (^ (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan2 (ship-y-velocity ship)
         (ship-x-velocity ship)))
```

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and do arithmetic. Those facts would not be considered part of the black box characteristics of the implementation of the ship type. The ship type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between ship and its callers. In fact, ship could have been written this way instead:

```
(defstruct (ship :conc-name)
  x-position
  y-position
  speed
  direction
  mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction ship))))
```

In this second implementation of the ship type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision. The caller has no idea which of the two ways the implementation uses; he just performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the ship structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides eq-ness) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the ship operations have different instance variables, but from the outside they have exactly the same operations.

One might ask: "But what if the caller evaluates **(aref ship 2)** and notices that he gets back the *x* velocity rather than the speed? Then he can tell which of the two implementations were used." This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised, the functions that are considered to be operations on the type of object. The contract from **ship** to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using **aref**. A caller who does so *is in error*; he is depending on something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, **ship** may get reimplemented overnight, and the code that does the **aref** will have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract specifies the interface between the two modules.

Unlike some other languages that provide abstract types, Zetalisp makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that the Lisp Machine is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the "system" programs and the "user" programs on the Lisp Machine; users are allowed to get into any part of the language system and change what they want to change. Another reason is the traditional MIT AI Lab philosophy that opposes "fascist" restrictions which impose on the user "for his own good". The user himself should decide what is good for him.

In summary: by defining a set of operations and making only a specific set of external entrypoints available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The important thing is not that they are functions—in Lisp everything is done with functions. The important thing is that we have defined a new conceptual operation and given it a name, rather than requiring anyone who wants to do the operation to write it out step-by-step. Thus we say **(ship-x-velocity s)** rather than **(aref s 2)**.

Often a few abstract operation functions are simple enough that it is desirable to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding*.) The compiler is directed to do this through use of macros, substs, or optimizers. **defstruct** arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the Lisp code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or whatever); even if you change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed programs that use it may need to be recompiled. This is something we sometimes accept for the sake of efficiency.

In the present implementation of flavors, which is discussed below, there is no such compiler incorporation of nonmodular knowledge into a program, except when the :ordered-instance-variables feature is used; see page 427, where this problem is explained further. If you don't use the :ordered-instance-variables feature, you don't have to worry about this.

21.3 Generic Operations

Suppose we think about the rest of the program that uses the ship abstraction. It may want to deal with other objects that are like ship's in that they are movable objects with mass, but unlike ships in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object the attributes apply to. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship needs to know the ship's attributes, and must therefore call ship-x-position and ship-y-velocity and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls ship-x-position, the second one would call meteor-x-position, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in most programming languages, including Zetalisp. The + (or plus) function accepts integers, floats, ratios and complex numbers, and perform an appropriate kind of addition, based on the data types of the objects being manipulated. In our example, we need a generic x-position operation that can be performed on either ship's, meteor's, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the *x* position of the object it is dealing with, it simply invokes the generic x-position operation on the object, and whatever type of object it has, the correct operation is performed, and the *x* position is returned.

Another terminology for the use of such generic operations has emerged from the Smalltalk language: performing a generic operation is called *sending a message*. The message consists of an operation name (a symbol) and arguments. The objects in the program are thought of as little people, who get sent messages and respond with answers (returned values). In the example above, the objects are sent x-position messages, to which they respond with their *x* position.

Sending a message is a way of invoking a function without specifying which function is to be called. Instead, the data determines the function to use. The caller specifies an operation name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke is found from this information.

The two data used to figure out which function to call are the *type* of the object, and the *name* of the operation. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the operation is data which are passed as arguments to the function, so the operation is the only part of the message used to find the function. Such a function is called a *method*. For example, if we send an **x-position** message to an object of type **ship**, then the function we find is "the **ship** type's **x-position** method". A method is a function that handles a specific operation on a specific kind of object; this method handles messages named **x-position** to objects of type **ship**.

In our new terminology: the orbit-calculating program finds the *x* position of the object it is working on by sending that object a message consisting of the operation **x-position** and no arguments. The returned value of the message is the *x* position of the object. If the object was of type **ship**, then the **ship** type's **x-position** method was invoked; if it was of type **meteor**, then the **meteor** type's **x-position** method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing: the same operation can mean different things depending on the type of the object.

21.4 Generic Operations in Lisp

How do we implement message passing in Lisp? Our convention is that objects that receive messages are always *functional* objects (that is, you can apply them to arguments). A message is sent to an object by calling that object as a function, passing the operation name as the first argument and the arguments of the message as the rest of the arguments. Operation names are represented by symbols; normally these symbols are in the keyword package (see chapter 27, page 636), since messages are a protocol for communication between different programs, which may reside in different packages. So if we have a variable **my-ship** whose value is an object of type **ship**, and we want to know its *x* position, we send it a message as follows:

```
(send my-ship :x-position)
```

To set the ship's *x* position to 3.0, we send it a message like this:

```
(send my-ship :set :x-position 3.0)
```

It should be stressed that no new features are added to Lisp for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as an operation name. The object must consider this operation name, find the function which is the method for that operation, and invoke that function.

send *object operation* &rest *arguments*

Sends *object* a message with operation and arguments as specified. Currently **send** is identical to **funcall**, but preferable when a message is being sent, just for clarity.

There are vague ideas of making *send* different from *funcall* if *object* is a symbol, list, number, or other object that does not normally handle messages when funcalled, but the meaning of this is not completely clear.

lexpr-send *object operation* &rest *arguments*

Currently **lexpr-send** is the same as *apply*.

This raises the question of how message receiving works. The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function. But an ordinary function will not do. We need something that can store the instance variables (the internal state) of the object. We need a function with internal state; that is, we need a coroutine.

Of the Zetalisp features presented so far, the most appropriate is the closure (see chapter 12, page 250). A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **selectq** form to dispatch on its first argument. (Actually, rather than using closures and a **selectq**, you would probably use entities (section 12.4, page 255) and **defselect** (page 236).)

While using closures (or entities) does work, it has several serious problems. The main problem is that in order to add a new operation to a system, it is necessary to modify a lot of code; you have to find all the types that understand that operation, and add a new clause to the **selectq**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system; the methods must be interleaved with the other operations for the type. Adding a new operation should only require *adding* Lisp code; it should not require *modifying* Lisp code.

The conventional way of making generic operations is to have a procedure for each operation, which has a big **selectq** for all the types; this means you have to modify code to add a type. The way described above is to have a procedure for each type, which has a big **selectq** for all the operations; this means you have to modify code to add an operation. Neither of these has the desired property that extending the system should only require adding code, rather than modifying code.

Closures (and entities) are also somewhat clumsy and crude. A far more streamlined, convenient, and powerful system for creating message-receiving objects exists; it is called the *flavor* mechanism. With flavors, you can add a new method simply by adding code, without modifying anything. Furthermore, many common and useful things are very easy to do with flavors. The rest of this chapter describes flavors.

21.5 Simple Use of Flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the defflavor special form, and methods of the flavor are created with the defmethod special form. New instances of a flavor are created with the make-instance function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the ship example above would be implemented.

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables)

(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
           (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan2 y-velocity x-velocity))
```

The code above creates a new flavor. The first subform of the defflavor is ship, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of the subforms are the body of the defflavor, and each one specifies an option about this flavor. In our example, there is only one option, namely :gettable-instance-variables. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the operation is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the operations :x-position, :y-position, and so on.

Each of the two defmethod forms adds a method to the flavor. The first one adds a handler to the flavor ship for the operation :speed. The second subform is the lambda-list, and the rest is the body of the function that handles the :speed operation. The body can refer to or set any instance variables of the flavor, just like variables bound by a containing let. When any instance of the ship flavor is invoked with a first argument of :direction, the body of the second defmethod is evaluated in an environment in which the instance variables of ship refer to the instance variables of this instance (the one to which the message was sent). So the arguments passed to cli:atan are the the velocity components of this particular ship. The result of cli:atan becomes the value returned by the :direction operation.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor has the five instance variables named in the defflavor form, and the seven methods we have seen (five that were automatically generated because of the :gettable-instance-variables option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the make-instance function. Here is how it could be used:

```
    (setq my-ship (make-instance 'ship))
```

This returns an object whose printed representation is #<SHIP 13731210>. (Of course, the value of the magic number will vary; it is just the object address in octal.) The argument to make-instance is the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily by putting the :settable-instance-variables option into the defflavor form. This option tells defflavor to generate methods for operation :set for first argument :x-position, :y-position, and so on; each such method takes one additional argument and sets the corresponding instance variable to that value. It also generates methods for the operations :set-x-position, :set-y-position and so on; each of these takes one argument and sets the corresponding variable.

Another option we can add to the defflavor is :inittable-instance-variables, which allows us to initialize the values of the instance variables when an instance is first created. :inittable-instance-variables does not create any methods; instead, it makes *initialization keywords* named :x-position, :y-position, etc., that can be used as init-option arguments to make-instance to initialize the corresponding instance variables. The list of init options is sometimes called the *init-plist* because it is like a property list.

Here is the improved defflavor:

```
    (defflavor ship (x-position y-position
                     x-velocity y-velocity mass)
                    ()
      :gettable-instance-variables
      :settable-instance-variables
      :inittable-instance-variables)
```

All we have to do is evaluate this new defflavor, and the existing flavor definition is updated and now includes the new methods and initialization options. In fact, the instance we generated a while ago now accepts the new operations! We can set the mass of the ship we created by evaluating

```
    (send my-ship :set-mass 3.0)
```

or

```
    (send my-ship :set :mass 3.0)
```

and the mass instance variable of my-ship is properly set to 3.0. Whether you use :set-mass or the general operation :set is a matter of style; :set is used by the expansion of (setf (send my-ship :mass) 3.0).

If you want to play around with flavors, it is useful to know that describe of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate (describe my-ship) at this point, the following would be printed:

```
#<SHIP 13731210>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:          void
        Y-POSITION:          void
        X-VELOCITY:          void
        Y-VELOCITY:          void
        MASS:                3.0
```

Now that the instance variables are *inittable*, we can create another ship and initialize some of the instance variables using the init-plist. Let's do that and describe the result:

```
(setq her-ship (make-instance 'ship :x-position 0.0
                                    :y-position 2.0
                                    :mass 3.5))
                => #<SHIP 13756521>


(describe her-ship)
#<SHIP 13756521>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:          0.0
        Y-POSITION:          2.0
        X-VELOCITY:          void
        Y-VELOCITY:          void
        MASS:                3.5
```

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)

(setq another-ship (make-instance 'ship :x-position 3.4))
 => #<SHIP 14563643>

(describe another-ship)
#<SHIP 14563643>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         3.4
        Y-POSITION:         0.0
        X-VELOCITY:         2.0
        Y-VELOCITY:         3.0
        MASS:               void
```

x-position was initialized explicitly, so the default was ignored. y-position was initialized from the default value, which was 0.0. The two velocity instance variables were initialized from their default values, which came from two global variables. mass was not explicitly initialized and did not have a default initialization, so it was left void.

There are many other options that can be used in **defflavor**, and the init options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

21.6 Mixing Flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to create a new type called meteor that would accept the same generic operations as ship, we could simply write another **defflavor** and two more **defmethod**'s that looked just like those of ship, and then meteors and ships would both accept the same operations. ship would have some more instance variables for holding attributes specific to ships and some more methods for operations that are not generic, but are only defined for ships; the same would be true of meteor.

However, this would be a a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name "flavors") comes from the ability to mix several flavors and get a new flavor. Since the functionality of ship and meteor partially overlap, we can take the common functionality and move it into its own flavor, which might be called moving-object. We would define moving-object the same way as we defined ship in the previous section. Then, ship and meteor could be defined like this:

```
(defflavor ship (engine-power number-of-passengers name)
                (moving-object)
   :gettable-instance-variables)

(defflavor meteor (percent-iron)
                  (moving-object)
   :inittable-instance-variables)
```

These defflavor forms use the second subform, which we ignored previously. The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on ship for a moment (analogous things are true of meteor), we see that it has exactly one component flavor: moving-object. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with meteor. By incorporating moving-object, the ship flavor acquires all of its instance variables, and so need not name them again. It also acquires all of moving-object's methods, too. So with the new definition, ship instances still implement the :x-velocity and :speed operations, with the same meaning as before. However, the :engine-power operation is also understood (and returns the value of the engine-power instance variable).

What we have done here is to take an abstract type, moving-object, and build two more specialized and powerful abstract types on top of it. Any ship or meteor can do anything a moving object can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called ship-with-passenger that was built on top of ship, and it would inherit all of moving-object's instance variables and methods as well as ship's instance variables and methods. Furthermore, the second subform of defflavor can be a list of several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term "components" to mean the immediate components (the ones listed in the defflavor), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes *before* the subtrees they head, ignoring any flavor that has been encountered previously somewhere else in the tree. For example, if flavor-1's immediate components are flavor-2 and flavor-3, and flavor-2's components are flavor-4 and flavor-5, and flavor-3's component was flavor-4, then the complete list of components of flavor-1 would be:

```
flavor-1, flavor-2, flavor-4, flavor-5, flavor-3
```

The flavors earlier in this list are the more specific, less basic ones; in our example, ship-with-

passengers would be first in the list, followed by **ship**, followed by **moving-object**. A flavor is always the first in the list of its own components. Notice that **flavor-4** does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; if there is a cycle in the directed graph, it does not cause a non-terminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both **flavor-2** and **flavor-3** have instance variables named **foo**, then **flavor-1** has an instance variable named **foo**, and any methods that refer to **foo** refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Typically, only one component ever sets the variable; the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each operation supported by the flavor. This function is constructed out of all the methods for that operation from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods. The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor **foo** and building a flavor **bar** on top of it, then you can override **foo**'s method for an operation by providing your own method. Your method will be called, and **foo**'s will never be called.

Simple overriding is often useful; for example, if you want to make a new flavor **bar** that is just like **foo** except that it reacts completely differently to a few operations. However, often you don't want to completely override the base flavor's (**foo**'s) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is "in charge" of the main business of handling the operation, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

daemon methods come in two kinds, *before* and *after*. There is a special syntax in **defmethod** for defining such methods. Here is an example of the syntax. To give the **ship** flavor an after-daemon method for the **:speed** operation, the following syntax would be used:

```
    (defmethod (ship :after :speed) () body)
```

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-daemons are called in the reverse order. In other words, if you build **bar** on top of **foo**, then **bar**'s before-daemons run before any of those in **foo**, and **bar**'s after-daemons run after any of those in **foo**.

The reason for this order is to keep the modularity order correct. If we create **flavor-1** built on **flavor-2**, then it should not matter what **flavor-2** is built out of. Our new before-daemons go before all methods of **flavor-2**, and our new after-daemons go after all methods of **flavor-2**. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of **defmethod** below.

To make this a bit more clear, let's consider a simple example that is easy to play with: the **:print-self** method. The Lisp printer (i.e. the **print** function; see section 23.1, page 506) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** operation is a stream (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the stream. In the **ship** example above, the reason that instances of the **ship** flavor printed the way they did is because the **ship** flavor was actually built on top of a very basic flavor called **vanilla-flavor**; this component is provided automatically by **defflavor**. It was **vanilla-flavor**'s **:print-self** method that was doing the printing. Now, if we give **ship** its own primary method for the **:print-self** operation, then that method completely takes over the job of printing; **vanilla-flavor**'s method will not be called at all. However, if we give **ship** a before-daemon method for the **:print-self** operation, then it will get invoked before the **vanilla-flavor** method, and so whatever it prints will appear before what **vanilla-flavor** prints. So we can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section; see section 21.11, page 433. **vanilla-flavor** and what it does for you are also explained later; see section 21.10, page 432.

21.7 Flavor Functions

defflavor *Macro*

A flavor is defined by a form

```
        (defflavor flavor-name (var1 var2...) (flav1 flav2...)
                   opt1 opt2...)
```

flavor-name is a symbol which serves to name this flavor. It is given an **si:flavor** property which is the internal data-structure containing the details of the flavor.

(type-of *obj***)**, where *obj* is an instance of the flavor named *flavor-name*, returns the symbol *flavor-name*. **(typep** *obj flavor-name***)** is **t** if *obj* is an instance of a flavor, one of whose components (possibly itself) is *flavor-name*.

var1, *var2*, etc. are the names of the instance-variables containing the local state for this flavor. A list of the name of an instance-variable and a default initialization form is also acceptable; the initialization form is evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable remains void.

flav1, *flav2*, etc. are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

opt1, *opt2*, etc. are options; each option may be either a keyword symbol or a list of a keyword symbol and arguments. The options to **defflavor** are described in section 21.8, page 424.

all-flavor-names *Variable*

A list of the names of all the flavors that have ever been **defflavor**'ed.

defmethod *Macro*

A method, that is, a function to handle a particular operation for instances of a particular flavor, is defined by a form such as

```
(defmethod (flavor-name method-type operation) lambda-list
  form1 form2...)
```

flavor-name is a symbol which is the name of the flavor which is to receive the method. *operation* is a keyword symbol which names the operation to be handled. *method-type* is a keyword symbol for the type of method; it is omitted when you are defining a primary method. For some method-types, additional information is expected. It comes after *operation*.

The meaning of *method-type* depends on what style of method combination is declared for this operation. For instance, if **:daemon** combination (the default style) is in use, method types **:before** and **:after** are allowed. See section 21.11, page 433 for a complete description of method types and the way methods are combined.

lambda-list describes the arguments and aux variables of the function; the first argument to the method, which is the operation name itself, is automatically handled and so is not included in the lambda-list. Note that methods may not have unevaluated **(&quote)** arguments; that is, they must be functions, not special forms. *form1*, *form2*, etc. are the function body; the value of the last form is returned.

The variant form

```
(defmethod (flavor-name operation) function)
```

where *function* is a symbol, says that *flavor-name*'s method for *operation* is *function*, a symbol which names a function. That function must take appropriate arguments; the first argument is the operation. When the function is called, **self** will be bound.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, an operation name, and a method type, there can only be one function (with the exception of **:case** methods; see page 437), so if you define a **:before** daemon method for the **foo** flavor to handle the **:bar** operation, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, operation or flavor.

The function spec for a method (see section 11.2, page 225) looks like:
```
        (:method flavor-name operation)   or
        (:method flavor-name method-type operation)   or
        (:method flavor-name method-type operation suboperation)
```
This is useful to know if you want to trace (page 738), breakon (page 741) or advise (page 742) a method, or if you want to poke around at the method function itself, e.g. disassemble it (see page 792).

make-instance *flavor-name init-option1 value1 init-option2 value2...*

Creates and returns an instance of the specified flavor. Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. An **:init** message is sent to the newly-created object with one argument, the init-plist. This is a disembodied property-list containing the init-options specified and those defaulted from the flavor's **:default-init-plist** (however, init keywords that simply initialize instance variables, and the corresponding values, may be absent when the :init methods are called). **make-instance** is an easy-to-call interface to **instantiate-flavor**, below.

If **:allow-other-keys** is used as an init keyword with a non-**nil** value, this error check is suppressed. Then unrecognized keywords are simply ignored. Example:
```
        (make-instance 'foo :lose 5 :allow-other-keys t)
```
specifies the init keyword **:lose**, but prevents an error should the keyword not be handled.

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords area*

This is an extended version of **make-instance**, giving you more features. Note that it takes the *init-plist* as an individual argument, rather than taking a rest argument of init options and values.

The *init-plist* argument must be a disembodied property list; **locf** of a rest argument is satisfactory. Beware! This property list can be modified; the properties from the default init plist are **putprop**'ed on if not already present, and some :init methods do explicit **putprop**'s onto the *init-plist*.

In the event that :init methods **remprop** properties already on the *init-plist* (as opposed to simply doing **get** and **putprop**), then the *init-plist* is **rplacd**'ed. This means that the actual supplied list of options is modified. It also means that **locf** of a rest argument does not work; the caller of **instantiate-flavor** must copy its rest argument (e.g. with **copylist**); this is because **rplacd** is not allowed on stack lists.

Do not use **nil** as the *init-plist* argument. This would mean to use the properties of the symbol **nil** as the init options. If your goal is to have no init options, you must provide a property list containing no properties, such as the list **(nil)**.

Here is the sequence of actions by which **instantiate-flavor** creates a new instance:

First, the specified flavor's instantiation flavor function (page 429), if it exists, is called to determine which flavor should actually be instantiated. If there is no instantiation flavor function, the specified flavor is instantiated.

If the flavor's method hash-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time or even invoke the compiler, but it happens only once for each time you define or redefine a particular flavor.

Next, the instance itself is created. If the *area* argument is specified, it is the number of an area in which to cons the instance; otherwise the flavor's instance area function is called to choose an area if there is one; otherwise, **default-cons-area** is used. See page 429.

Then the initial values of the instance variables are computed. If an instance variable is declared inittable, and a keyword with the same spelling as its name appears in *init-plist*, the property for that keyword is used as the initial value.

Otherwise, if the default init plist specifies such a property, it is evaluated and the value is used. Otherwise, if the flavor definition specifies a default initialization form, it is evaluated and the value is used. The initialization form may not refer to any instance variables. It can find the new instance in **self** but should not invoke any operations on it and should not refer directly to any instance variables. It can get at instance variables using accessor macros created by the **:outside-accessible-instance-variables** option (page 427) or the function **symeval-in-instance** (page 423).

If an instance variable does not get initialized either of these ways it is left void; an **:init** method may initialize it (see below).

All remaining keywords and values specified in the **:default-init-plist** option to **defflavor**, that do not initialize instance variables and are not overridden by anything explicitly specified in *init-plist* are then merged into *init-plist* using **putprop**. The default init plist of the instantiated flavor is considered first, followed by those of all the component flavors in the standard order. See page 425.

Then keywords appearing in the *init-plist* but not defined with the **:init-keywords** option or the **:inittable-instance-variables** option for some component flavor are collected. If the **:allow-other-keys** option is specified with a non-**nil** value (either in the original *init-plist* argument or by some default init plist) then these *unhandled* keywords are ignored. If the *return-unhandled-keywords* argument is non-**nil**, a list of these keywords is returned as the second value of **instantiate-flavor**. Otherwise, an error is signaled if any unrecognized init keywords are present.

If the *send-init-message-p* argument is supplied and non-**nil**, an **:init** message is sent to the newly-created instance, with one argument, the *init-plist*. **get** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an **:init** method by defining a daemon.

The **:init** methods should not look on the *init-plist* for keywords that simply initialize instance variables (that is, keywords defined with **:inittable-instance-variables** rather than **:init-keywords**). The corresponding instance variables are already set up when the **:init** methods are called, and sometimes the keywords and their values may actually be missing from the *init-plist* if it is more efficient not to put them on. To avoid problems, always refer to the instance variables themselves rather than looking for the init keywords that initialize them.

:init *init-plist* *Operation on all flavor instances*

This operation is implemented on all flavor instances. Its purpose is to examine the init keywords and perform whatever initializations are appropriate. *init-plist* is the argument that was given to **instantiate-flavor**, and may be passed directly to **get** to examine the value of any particular init option.

The default definition of this operation does nothing. However, many flavors add **:before** and **:after** daemons to it.

instancep *object*

Returns **t** if **object** is an instance. This is equivalent to **(typep** *object* **'instance)**.

defwrapper *Macro*

This is hairy and if you don't understand it you should skip it.

Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro that expands into code that is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the **:foo** operation on flavor **bar**, which takes two arguments, and you have a **lock-frobboz** special-form that knows how to lock the lock (presumably it generates an **unwind-protect**). **lock-frobboz** needs to see the first argument to the operation; perhaps that tells it what sort of operation is going to be performed (read or write).

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  '(lock-frobboz (self arg1)
     . ,body))
```

The use of the **body** macro-argument prevents the macro defined by **defwrapper** from knowing the exact implementation and allows several **defwrapper**'s from different flavors to be combined properly.

Note well that the argument variables, **arg1** and **arg2**, are not referenced with commas before them. These may look like **defmacro** "argument" variables, but they are not. Those variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done; rather the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a **:before** daemon, but found that if the argument was **nil** you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as

```
(defwrapper (bar :foo) ((arg1) . body)
  '(cond ((null arg1))
         (t (print "About to do :FOO")
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular operation; perhaps the :after daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is bound during the processing of the operation and used free by the methods.

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  '(let ((*communication* nil))
     . ,body))
```

Similarly you might want a wrapper that puts a **catch** around the processing of an operation so that any one of the methods could throw out in the event of an unexpected condition.

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. 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.

:around methods can do some of the same things that wrappers can. See page 439. If one flavor defines both a wrapper and an :around method for the same operation, the :around method is executed inside the wrapper.

By careful about inserting the body into an internal lambda-expression within the wrapper's code. Doing so interacts with the internals of the flavor system and requires knowledge of things not documented in the manual in order to work properly. It is much simpler to use an :around method instead.

undefmethod (*flavor* [*type*] *operation* [*suboperation*]) *Macro*

```
(undefmethod (flavor :before :operation))
```

removes the method created by

```
(defmethod (flavor :before :operation) (args) ...)
```

To remove a wrapper, use **undefmethod** with :wrapper as the method type.

undefmethod is simply an interface to **fundefine** (see page 241) that accepts the same syntax as **defmethod**.

If a file that used to contain a method definition is reloaded and if that method no longer seems to have a definition in the file, the user is asked whether to **undefmethod** that method. This may be important to enable the modified program to inherit the methods it is supposed to inherit. If the method in question has been redefined by some other file, this is not done, the assumption being that the definition was merely moved.

undefflavor *flavor*

Undefines flavor *flavor*. All methods of the flavor are lost. *flavor* and all flavors that depend on it are no longer valid to instantiate.

If instances of the discarded definition exist, they continue to use that definition.

self *Variable*

When a message is sent to an object, the variable **self** is automatically bound to that object, for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables).

funcall-self *operation arguments...*
lexpr-funcall-self *operation arguments... list-of-arguments*

funcall-self is nearly equivalent to **funcall** with **self** as the first argument. **funcall-self** used to be faster, but now **funcall** of **self** is just as fast. Therefore, **funcall-self** is obsolete. It should be replaced with **funcall** or **send** of **self**.

Likewise, **lexpr-funcall-self** should be replaced with use of **lexpr-send** to **self**.

funcall-with-mapping-table *function mapping-table* &rest *arguments*

Applies *function* to *arguments* with **sys:self-mapping-table** bound to *mapping-table*. This is faster than binding the variable yourself and doing an ordinary **funcall**, because the system assumes that the mapping table you specify is the correct one for *function* to be run with. However, if you pass the wrong mapping table, incorrect execution will take place.

This function is used in the code for combined methods and is also useful for the user in :around methods (see page 439).

lexpr-funcall-with-mapping-table *function mapping-table* &rest *arguments*

Applies *function* to *arguments* using **lexpr-funcall**, with **sys:self-mapping-table** bound to *mapping-table*.

declare-flavor-instance-variables (*flavor*) *body...* *Macro*

Sometimes it is useful to have a function which is not itself a method, but which is to be called by methods and wants to be able to access the instance variables of the object **self**. The form

```
    (declare-flavor-instance-variables (flavor-name)
      (defun function args body...))
```

surrounds the function definition with a peculiar kind of declaration which makes the instance variables of flavor *flavor-name* accessible by name. Any kind of function definition is allowed; it does not have to use **defun** per se.

If you call such a function when **self**'s value is an instance whose flavor does not include *flavor-name* as a component, it is an error.

Cleaner than using **declare-flavor-instance-variables**, because it does not involve putting anything around the function definition, is using a local declaration. Put **(declare (:self-flavor** *flavorname***))** as the first expression in the body of the function. For example:

```
(defun foo (a b)
  (declare (:self-flavor myobject))
  (+ a (* b speed)))
```

(where **speed** is an instance variable of the flavor **myobject**) is equivalent to

```
(declare-flavor-instance-variables (myobject)
(defun foo (a b)
  (+ a (* b speed))))
```

with-self-variables-bound *body...* *Special form*

Within the body of this special form, all of **self**'s instance variables are bound as specials to the values inside **self**. (Normally this is true only of those instance variables that are specified in **:special-instance-variables** when **self**'s flavor was defined.) As a result, inside the body you can use **set**, **boundp** and **symeval**, etc., freely on the instance variables of **self**.

recompile-flavor *flavor-name* &optional *single-operation* (*use-old-combined-methods* **t**) (*do-dependents* **t**)

Updates the internal data of the flavor and any flavors that depend on it. If *single-operation* is supplied non-**nil**, only the methods for that operation are changed. The system does this when you define a new method that did not previously exist. If *use-old-combined-methods* is **t**, then the existing combined method functions are used if possible. New ones are generated only if the set of methods to be called has changed. This is the default. If *use-old-combined-methods* is **nil**, automatically-generated functions to call multiple methods or to contain code generated by wrappers are regenerated unconditionally. If *do-dependents* is **nil**, only the specific flavor you specified is recompiled. Normally all flavors that depend on it are also recompiled.

recompile-flavor affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins (see page 431).

si:*dont-recompile-flavors* *Variable*

If this variable is non-**nil**, automatic recompilation of combined methods is turned off.

If you wish to make several changes each of which will cause recompilation of the same combined methods, you can use this variable to speed things up by making the recompilations happen only once. Set the variable to **t**, make your changes, and then set the variable back to **nil**. Then use **recompile-flavor** to recompile whichever combined methods need it. For example:

```
(setq si:*dont-recompile-flavors* t)
(undefmethod (tv:sheet :after :bar))
(defmethod (tv:sheet :before :bar) ...)
(setq si:*dont-recompile-flavors* nil)
(recompile-flavor 'tv:sheet :bar)
```

tv:sheet has very many dependents; **recompile-flavor** even once takes painfully long. It's nice to avoid spending the time twice.

compile-flavor-methods *flavor...* *Macro*

The form (**compile-flavor-methods** *flavor-name-1 flavor-name-2...*), placed in a file to be compiled, directs the compiler to include the automatically-generated combined methods for the named flavors in the resulting QFASL file, provided all of the necessary flavor definitions have been made. Furthermore, all internal data structures needed to instantiate the flavor will be computed when the QFASL file is loaded rather than waiting until the first attempt to instantiate it.

This means that the combined methods get compiled at compile time and the data structures get generated at load time, rather than both things happening at run time. This is a very good thing, since if the the compiler must be invoked at run time, the program will be slow the first time it is run. (The compiler must be called in any case if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method.)

You should only use **compile-flavor-methods** for flavors that are going to be instantiated. For a flavor that is never to be instantiated (that is, a flavor that only serves to be a component of other flavors that actually do get instantiated), it is a complete waste of time, except in the unusual case where those other flavors can all inherit the combined methods of this flavor instead of each one having its own copy of a combined method which happens to be identical to the others. In this unusual case, you should use the :abstract-flavor option in **defflavor** (page 428).

The **compile-flavor-methods** forms should be compiled after all of the information needed to create the combined methods is available. You should put these forms after all of the definitions of all relevant flavors, wrappers, and methods of all components of the flavors mentioned.

The methods used by **compile-flavor-methods** to form the combined methods that go in the QFASL file are all those present in the file being compiled and all those defined in the Lisp world.

When a **compile-flavor-methods** form is seen by the interpreter, the combined methods are compiled and the internal data structures are generated.

get-handler-for *object operation*

Given an object and an operation, this returns the object's method for that operation, or nil if it has none. When *object* is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method. If you get back a combined method, you can use the **Meta-X List Combined Methods** editor command (page 444) to find out what it does.

This is related to the **:handler** function spec (see section 11.2, page 223).

It is preferable to use the generic operation **:get-handler-for**.

flavor-allows-init-keyword-p *flavor-name keyword*
Returns non-nil if the flavor named *flavor-name* allows *keyword* in the init options when it is instantiated, or nil if it does not. The non-nil value is the name of the component flavor that contributes the support of that keyword.

si:flavor-all-allowed-init-keywords *flavor-name*
Returns a list of all the init keywords that may be used in instantiating *flavor-name*.

symeval-in-instance *instance symbol* &optional *no-error-p*
Returns the value of the instance variable *symbol* inside *instance*. If there is no such instance variable, an error is signaled, unless *no-error-p* is non-nil in which case nil is returned.

set-in-instance *instance symbol value*
Sets the value of the instance variable *symbol* inside *instance* to *value*. If there is no such instance variable, an error is signaled.

locate-in-instance *instance symbol*
Returns a locative pointer to the cell inside *instance* which holds the value of the instance variable named *symbol*.

describe-flavor *flavor-name*
Prints descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined list of component flavors; this list is what is printed after the phrase 'and directly or indirectly depends on'.

si:*flavor-compilations* *Variable*
Contains a history of when the flavor mechanism invoked the compiler. It is a list; elements toward the front of the list represent more recent compilations. Elements are typically of the form

(*function-spec pathname*)

where the function spec starts with :method and has a method type of :combined.

You may setq this variable to nil at any time; for instance before loading some files that you suspect may have missing or obsolete compile-flavor-methods in them.

sys:unclaimed-message (error) *Condition*
This condition is signaled whenever a flavor instance is sent a message whose operation it does not handle. The condition instance supports these operations:

:object	The flavor instance that received the message.
:operation	The operation that was not handled.
:arguments	The list of arguments to that operation

21.8 Defflavor Options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. Each option can be written in two forms; either the keyword by itself, or a list of the keyword and arguments to that keyword.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the component flavors, just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signaled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables
: Enables automatic generation of methods for getting the values of instance variables. The operation name is the name of the variable, in the keyword package (i.e. it has a colon in front of it).

  Note that there is nothing special about these methods; you could easily define them yourself. This option generates them automatically to save you the trouble of writing out a lot of very simple method definitions. (The same is true of methods defined by the :settable-instance-variables option.) If you define a method for the same operation name as one of the automatically generated methods, the explicit definition overrides the automatic one.

:settable-instance-variables
: Enables automatic generation of methods for setting the values of instance variables. The operation name is ':set-' followed by the name of the variable. All settable instance variables are also automatically made gettable and inittable. (See the note in the description of the :gettable-instance-variables option, above.)

  In addition, :case methods are generated for the :set operation with suboperations taken from the names of the variables, so that :set can be used to set them.

:inittable-instance-variables
: The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *inittable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**.

:special-instance-variables
: The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, will be bound dynamically when handling messages. (By default, instance variables are bound lexically with the scope being the method.) You must do this to any instance variables that you wish to be accessible through **symeval**, **set**, **boundp** and **makunbound**, since they see only dynamic bindings.

This should also be done for any instance variables that are declared globally special. If you omit this, the flavor system does it for you automatically when you instantiate the flavor, and gives you a warning to remind you to fix the **defflavor**.

:init-keywords

The arguments are declared to be valid keywords to use in **instantiate-flavor** when creating an instance of this flavor (or any flavor containing it). The system uses this for error-checking: before the system sends the **:init** message, it makes sure that all the keywords in the init-plist are either inittable instance variables or elements of this list. If any is not recognized, an error is signaled. When you write a **:init** method that accepts some keywords, they should be listed in the **:init-keywords** option of the flavor.

If **:allow-other-keys** is used as an init keyword with a non-**nil** value, this error check is suppressed. Then unrecognized keywords are simply ignored.

:default-init-plist

The arguments are alternating keywords and value forms, like a property list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

```
(:default-init-plist :frob-array
                     (make-array 100))
```

would provide a default "frob array" for any instance for which the user did not provide one explicitly.

```
(:default-init-plist :allow-other-keys t)
```

prevents errors for unhandled init keywords in all instantiation of this flavor and other flavors that depend on it.

:required-init-keywords

The arguments are init keywords which are to be required each time this flavor (or any flavor containing it) is instantiated. An error is signaled if any required init keyword is missing.

:required-instance-variables

Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those that checks the spelling of its arguments in the way described at the start of this section (if it did, it would be useless).

Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and accepts responsibility for 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

The arguments are names of operations that any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for one of these operations. Typically this option appears in the **defflavor** for a base flavor (see page 431). Usually this is used when a base flavor does a **(send self ...)** to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message to be detected when the flavor instantiated or when **compile-flavor-methods** is done, rather than when the missing operation is used.

:required-flavors

The arguments are names of flavors that any flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever does specify them as components. The purpose of declaring a flavor to be required is to allow instance variables declared by that flavor to be accessed. It also provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components signals an error. Compare this with **:required-methods** and **:required-instance-variables.**

For an example of the use of required flavors, consider the **ship** example given earlier, and suppose we want to define a **relativity-mixin** which increases the mass dependent on the speed. We might write,

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (// mass (sqrt (- 1 (^ (// (send self :speed)
                             *speed-of-light*)
                         2)))))
```

but this would lose because any flavor that had **relativity-mixin** as a component would get **moving-object** right after it in its component list. As a base flavor, **moving-object** should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. **relativity-mixin** has no business changing the order in which flavors are combined, which should be under the control of its caller. For example,

```
(defflavor starship ()
           (relativity-mixin long-distance-mixin ship))
```

puts **moving-object** last (inheriting it from **ship**).

So instead of the definition above we write,

```
(defflavor relativity-mixin () ()
        (:required-flavors moving-object))
```

which allows **relativity-mixin**'s methods to access **moving-object** instance variables such as **mass** (the rest mass), but does not specify any place for **moving-object** in the list of components.

It is very common to specify the *base flavor* of a mixin with the **:required-flavors** option in this way.

:included-flavors

The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the **defflavor** is that when component flavors are combined, if an included flavor is not specified as a normal component, it is inserted into the list of components immediately after the last component to include it. Thus included flavors act like defaults. The important thing is that if an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where the flavor that includes it appears in the list.

:included-flavors and **:required-flavors** are used in similar ways; it would have been reasonable to use **:included-flavors** in the **relativity-mixin** example above. The difference is that when a flavor is required but not given as a normal component, an error is signaled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a reasonable place.

:no-vanilla-flavor
Normally when a flavor is instantiated, the special flavor **si:vanilla-flavor** is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard operations which all objects are supposed to understand. These include **:print-self**, **:describe**, **:which-operations**, and several other operations. See section 21.10, page 432.

If any component of a flavor specifies the **:no-vanilla-flavor** option, then **si:vanilla-flavor** is not included in that flavor. This option should not be used casually.

:default-handler
The argument is the name of a function that is to be called to handle any operation for which there is no method. Its arguments are the arguments of the **send** which invoked the operation, including the operation name as the first argument. Whatever values the default handler returns are the values of the operation.

Default handlers can be inherited from component flavors. If a flavor has no default handler, any operation for which there is no method signals a **sys:unclaimed-message** error.

:ordered-instance-variables
This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables that are specially known about by microcode, and also in connection with the **:outside-accessible-instance-variables** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

Removing any of the **:ordered-instance-variables**, or changing their positions in the list, requires that you recompile all methods that use any of the affected instance variables.

:outside-accessible-instance-variables
The arguments are instance variables which are to be accessible from outside of this flavor's methods. A macro (actually a subst) is defined which takes an object of this flavor as an argument and returns the value of the instance variable; **setf** may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are

similar to the accessor macros created by **defstruct** (see chapter 20, page 372.)

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances, via the **:ordered-instance-variables** option.

If the variable is not ordered, the position of its value cell in the instance must be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error is signaled if the argument to the accessor macro is not an instance or is an instance that does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

If the variable is ordered, the compiler compiles a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only three or four times slower than **car**. The only error-checking performed is to make sure that the argument is really an instance and is really big enough to contain that slot. There is no check that the accessed slot really belongs to an instance variable of the appropriate name.

:accessor-prefix

Normally the accessor macro created by the **:outside-accessible-instance-variables** option to access the flavor *f*'s instance variable *v* is named *f-v*. Specifying **(:accessor-prefix get$)** causes it to be named **get$***v* instead.

:alias-flavor

Marks this flavor as being an alias for another flavor. This flavor should have only one component, which is the flavor it is an alias for, and no instance variables or other options. No methods should be defined for it.

The effect of the **:alias-flavor** option is that an attempt to instantiate this flavor actually produces an instance of the other flavor. Without this option, it would make an instance of this flavor, which might behave identically to an instance of the other flavor. **:alias-flavor** eliminates the need for separate mapping tables, method tables, etc. for this flavor, which becomes truly just another name for its component flavor.

The alias flavor and its base flavor are also equivalent when used as an argument of **subtypep** or as the second argument of **typep**; however, if the alias status of a flavor is changed, you must recompile any code which uses it as the second argument to **typep** in order for such code to function.

:alias-flavor is mainly useful for changing a flavor's name gracefully.

:abstract-flavor

This option marks the flavor as one that is not supposed to be instantiated (that is, is supposed to be used only as a component of other flavors). An attempt to instantiate the flavor signals an error.

It is sometimes useful to do **compile-flavor-methods** on a flavor that is not going to be instantiated, if the combined methods for this flavor will be inherited and shared by many others. **:abstract-flavor** tells **compile-flavor-methods** not to complain about missing required flavors, methods or instance variables. Presumably the flavors that depend on this one and actually are instantiated will supply what is lacking.

:method-combination

Specifies the method combination style to be used for certain operations. Each argument to this option is a list (*style order operation1 operation2...*). *operation1*, *operation2*, etc. are names of operations whose methods are to be combined in the declared fashion. *style* is a keyword that specifies a style of combination; see section 21.11, page 433. *order* is a keyword whose interpretation is up to *style*; typically it is either **:base-flavor-first** or **:base-flavor-last**.

Any component of a flavor may specify the type of method combination to be used for a particular operation. If no component specifies a style of method combination, then the default style is used, namely **:daemon**. If more than one component of a flavor specifies the combination style for a given operation, then they must agree on the specification, or else an error is signaled.

:instance-area-function

The argument is the name of a function to be used when this flavor is instantiated, to determine which area to create the new instance in. Use a function name rather than an explicit lambda expression.

```
(:instance-area-function function-name)
```

When the instance area function is called, it is given the init plist as an argument, and should return an area number or **nil** to use the default. Init keyword values can be accessed using **get** on the init plist.

Instance area functions can be inherited from component flavors. If a flavor does not have or inherit an instance area function, its instances are created in **default-cons-area**.

:instantiation-flavor-function

You can define a flavor **foo** so that, when you try to instantiate it, it calls a function to decide what flavor it should really instantiate (not necessarily **foo**). This is done by giving **foo** an instantiation flavor function:

```
(:instantiation-flavor-function function-name)
```

When **(make-instance 'foo** *keyword-args...***)** is done, the instantiation flavor function is called with two arguments: the flavor name specified (**foo** in this case) and the init plist (the list of keyword args). It should return the name of the flavor that should actually be instantiated.

Note that the instantiation flavor function applies only to the flavor it is specified for. It is not inherited by dependent flavors.

:run-time-alternatives
:mixture

A run-time-alternative flavor defines a collection of similar flavors, all built on the same base flavor but having various mixins as well. Instantiation chooses a flavor of the collection at run time based on the init keywords specified, using an automatically generated instantiation flavor function.

A simple example would be

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
     (:big big-foo-mixin))
  (:init-keywords :big))
```

Then (make-instance 'foo :big t) makes an instance of a flavor whose components are big-foo-mixin as well as foo. But (make-instance 'foo) or (make-instance 'foo :big nil) makes an instance of foo itself. The clause (:big big-foo-mixin) in the :run-time-alternatives says to incorporate big-foo-mixin if :big's value is t, but not if it is nil.

There may be several clauses in the :run-time-alternatives. Each one is processed independently. Thus, two keywords :big and :wide could independently control two mixins, giving four possibilities.

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
     (:big big-foo-mixin)
     (:wide wide-foo-mixin))
  (:init-keywords :big))
```

It is possible to test for values other than t and nil. The clause

```
(:size (:big big-foo-mixin)
       (:small small-foo-mixin)
       (nil nil))
```

allows the value for the keyword :size to be :big, :small or nil (or omitted). If it is nil or omitted, no mixin is used (that's what the second nil means). If it is :big or :small, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values to check for can be anything, but eq is used to compare them.

The value of one keyword can control the interpretation of others by nesting clauses within clauses. If an alternative has more than two elements, the additional elements are subclauses which are considered only if that alternative is selected. For example, the clause

```
(:etherial (t etherial-mixin)
           (nil nil
                (:size (:big big-foo-mixin)
                       (:small small-foo-mixin)
                       (nil nil))))
```

says to consider the :size keyword only if :etherial is nil.

:mixture is synonymous with :run-time-alternatives. It exists for compatibility with Symbolics systems.

:documentation

Specifies the documentation string for the flavor definition, which is made accessible through **(documentation *flavorname* 'flavor)**.

This documentation can be viewed with the **describe-flavor** function (see page 423) or the editor's **Meta-X Describe Flavor** command (see page 443).

Previously this option expected two arguments, a keyword and a string. The keyword was intended to classify the flavor as a base flavor, mixin or combination. But no way was found for this classification to serve a useful purpose. Keyword are still accepted but no longer recommended for use.

21.9 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 a component. 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 operations, **: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 are not instantiatable and merely serve as a base upon which to build other flavors. The base flavor for the *foo* family is often named **basic-*foo***.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies should be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named ***mumble*-mixin**.

If you are writing a program that uses someone else's facility to do something, using that facility's flavors and methods, your program may still define its own flavors, in a simple way. The facility provides a base flavor and a set of mixins: the caller can combine these in various ways depending on exactly what it wants, since the facility probably does not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its **:default-init-plist** (see page 425) to select options of its component flavors and you can define one or two methods to customize it "just a little".

21.10 Vanilla Flavor

The operations described in this section are a standard protocol, which all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless the user specifically tells it not to do so. These methods are associated with the flavor **si:vanilla-flavor**:

si:vanilla-flavor *Flavor*

Unless you specify otherwise (with the :no-vanilla-flavor option to **defflavor**), every flavor includes the "vanilla" flavor, which has no instance variables but provides some basic useful methods.

:print-self *stream prindepth escape-p* *Operation*

The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether escaping is enabled (a copy of the value of ***print-escape***; see page 514). **si:vanilla-flavor** ignores the last two arguments and prints something like #<*flavor-name octal-address*>. The *flavor-name* tells you what type of object it is and the *octal-address* allows you to tell different objects apart (provided the garbage collector doesn't move them behind your back).

:describe *Operation*

The object should describe itself, printing a description onto the ***standard-output*** stream. The **describe** function sends this message when it encounters an instance. **si:vanilla-flavor** outputs in a reasonable format the object, the name of its flavor, and the names and values of its instance-variables.

:set *keyword value* *Operation*

The object should set the internal value specified by *keyword* to the new value *value*. For flavor instances, the :set operation uses **:case** method combination, and a method is generated automatically to set each settable instance variable, with *keyword* being the variable's name as a keyword.

:which-operations *Operation*

The object should return a list of the operations it can handle. **si:vanilla-flavor** generates the list once per flavor and remembers it, minimizing consing and compute-time. If the set of operations handled is changed, this list is regenerated the next time someone asks for it.

:operation-handled-p *operation* *Operation*

operation is an operation name. The object should return **t** if it has a handler for the specified operation, **nil** if it does not.

:get-handler-for *operation* *Operation*

operation is an operation name. The object should return the method it uses to handle *operation*. If it has no handler for that operation, it should return **nil**. This is like the **get-handler-for** function (see page 422), but, of course, you can use it only on objects known to accept messages.

:send-if-handles *operation* &rest *arguments* *Operation*

operation is an operation name and *arguments* is a list of arguments for the operation. If the object handles the operation, it should send itself a message with that operation and arguments, and return whatever values that message returns. If it doesn't handle the operation it should just return **nil**.

:eval-inside-yourself *form* *Operation*

The argument is a form that is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is intended to be used mainly for debugging.

:funcall-inside-yourself *function* &rest *args* *Operation*

function is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is a way of allowing callers to provide actions to be performed in an environment set up by the instance.

:break *Operation*

break is called in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables.

21.11 Method Combination

When a flavor has or inherits more than one method for an operation, they must be called in a specific sequence. The flavor system creates a function called a *combined method* which calls all the user-specified methods in the proper order. Invocation of the operation actually calls the combined method, which is responsible for calling the others.

For example, if the flavor **foo** has components and methods as follows:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))

(defmethod (foo :before :hack) ...)
(defmethod (foo :after :hack) ...)

(defmethod (foo-mixin :before :hack) ...)
(defmethod (foo-mixin :after :hack) ...)

(defmethod (bar-mixin :before :hack) ...)
(defmethod (bar-mixin :hack) ...)

(defmethod (foo-base :hack) ...)
(defmethod (foo-base :after :hack) ...)
```

then the combined method generated looks like this (ignoring many important details not related to this issue):

```
(defmethod (foo :combined :hack) (&rest args)
  (apply #'(:method foo :before :hack) args)
  (apply #'(:method foo-mixin :before :hack) args)
  (apply #'(:method bar-mixin :before :hack) args)
  (multiple-value-prog1
      (apply #'(:method bar-mixin :hack) args)
    (apply #'(:method foo-base :after :hack) args)
    (apply #'(:method foo-mixin :after :hack) args)
    (apply #'(:method foo :after :hack) args))).
```

This example shows the default style of method combination, the one described in the introductory parts of this chapter, called :daemon combination. Each style of method combination defines which *method types* it allows, and what they mean. :daemon combination accepts method types :before and :after, in addition to *untyped* methods; then it creates a combined method which calls all the :before methods, only one of the untyped methods, and then all the :after methods, returning the value of the untyped method. The combined method is constructed by a function much like a macro's expander function, and the precise technique used to create the combined method is what gives :before and :after their meaning.

Note that the :before methods are called in the order foo, foo-mixin, bar-mixin and foo-base. (foo-base does not have a :before method, but if it had one that one would be last.) This is the standard ordering of the components of the flavor foo (see page 412); since it puts the base flavor last, it is called :base-flavor-last ordering. The :after methods are called in the opposite order, in which the base flavor comes first. This is called :base-flavor-first ordering.

Only one of the untyped methods is used; it is the one that comes first in :base-flavor-last ordering. An untyped method used in this way is called a *primary* method.

Other styles of method combination define their own method types and have their own ways of combining them. Use of another style of method combination is requested with the :method-combination option to defflavor (see page 429). Here is an example which uses :list method combination, a style of combination that allows :list methods and untyped methods:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))
(defflavor foo-base () ()
  (:method-combination (:list :base-flavor-last :win)))

(defmethod (foo :list :win) ...)
(defmethod (foo :win) ...)

(defmethod (foo-mixin :list :win) ...)

(defmethod (bar-mixin :list :win) ...)
(defmethod (bar-mixin :win) ...)

(defmethod (foo-base :win) ...)
```

yielding the combined method

```
(defmethod (foo :combined :win) (&rest args)
  (list
    (apply #'(:method foo :list :win) args)
    (apply #'(:method foo-mixin :list :win) args)
    (apply #'(:method bar-mixin :list :win) args)
    (apply #'(:method foo :win) args)
    (apply #'(:method bar-mixin :win) args)
    (apply #'(:method foo-base :win) args)))
```

The **:method-combination** option in the **defflavor** for **foo-base** causes **:list** method combination to be used for the **:win** operation on all flavors that have **foo-base** as a component, including **foo**. The result is a combined method which calls all the methods, including all the untyped methods rather than just one, and makes a list of the values they return. All the **:list** methods are called first, followed by all the untyped methods; and within each type, the **:base-flavor-last** ordering is used as specified. If the **:method-combination** option said **:base-flavor-first**, the relative order of the **:list** methods would be reversed, and so would the untyped methods, but the **:list** methods would still be called before the untyped ones. **:base-flavor-last** is more often right, since it means that **foo**'s own methods are called first and **si:vanilla-flavor**'s methods (if it has any) are called last.

A few specific method types, such as **:default** and **:around**, have standard meanings independent of the style of method combination, and can be used with any style. They are described in a table below.

Here are the standardly defined method combination styles.

:daemon The default style of method combination. All the **:before** methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the **:after** methods are called. The value returned is the value of the primary method.

:daemon-with-or
Like the **:daemon** method combination style, except that the primary method is

wrapped in an :or special form with all :or methods. Multiple values can be returned from the primary method, but not from the :or methods (as in the **or** special form). This produces code like the following in combined methods:

```
(progn (foo-before-method)
       (multiple-value-prog1
         (or (foo-or-method)
             (foo-primary-method))
        (foo-after-method)))
```

This is useful primarily for flavors in which a mixin introduces an alternative to the primary method. Each :or method gets a chance to run before the primary method and to decide whether the primary method should be run or not; if any :or method returns a non-nil value, the primary method is not run (nor are the rest of the :or methods). Note that the ordering of the combination of the :or methods is controlled by the *order* keyword in the :method-combination option.

:daemon-with-and
: Like :daemon-with-or except that it combines :and methods in an **and** special form. The primary method is run only if all of the :and methods return non-nil values.

:daemon-with-override
: Like the :daemon method combination style, except an **or** special form is wrapped around the entire combined method with all :override typed methods before the combined method. This differs from :daemon-with-or in that the :before and :after daemons are run only if *none* of the :override methods returns non-nil. The combined method looks something like this:

```
(or (foo-override-method)
    (progn (foo-before-method)
           (foo-primary-method)
           (foo-after-method)))
```

:progn — Calls all the methods inside a **progn** special form. Only untyped and :progn methods are allowed. The combined method calls all the :progn methods and then all the untyped methods. The result of the combined method is whatever the last of the methods returns.

:or — Calls all the methods inside an **or** special form. This means that each of the methods is called in turn. Only untyped methods and :or methods are allowed; the :or methods are called first. If a method returns a non-nil value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it can return nil, and the next method gets a chance to try.

:and — Calls all the methods inside an **and** special form. Only untyped methods and :and methods are allowed. The basic idea is much like :or; see above.

:append — Calls all the methods and appends the values together. Only untyped methods and :append methods are allowed; the :append methods are called first.

:nconc — Calls all the methods and nconc's the values together. Only untyped methods and :nconc methods are allowed, etc.

:list — Calls all the methods and returns a list of their returned values. Only untyped methods and :list methods are allowed, etc.

:inverse-list — Calls each method with one argument; these arguments are successive elements of the list that is the sole argument to the operation. Returns no particular value. Only untyped methods and :inverse-list methods are allowed, etc.

If the result of a :list-combined operation is sent back with an :inverse-list-combined operation, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

:pass-on — Calls each method on the values returned by the preceeding one. The values returned by the combined method are those of the outermost call. The format of the declaration in the defflavor is:

```
(:method-combination (:pass-on (ordering . arglist))
                                  . operation-names)
```

where *ordering* is :base-flavor-first or :base-flavor-last. *arglist* may include the &aux and &optional keywords.

Only untyped methods and :pass-on methods are allowed. The :pass-on methods are called first.

:case — With :case method combination, the combined method automatically does a selectq dispatch on the first argument of the operation, known as the *suboperation*. Methods of type :case can be used, and each one specifies one suboperation that it applies to. If no :case method matches the suboperation, the primary method, if any, is called.

Example:

```
(defflavor foo (a b) ()
  (:method-combination (:case :base-flavor-last :win)))
```

This method handles (send a-foo :win :a):

```
(defmethod (foo :case :win :a) ()
  a)
```

This method handles (send a-foo :win :a*b):

```
(defmethod (foo :case :win :a*b) ()
  (* a b))
```

This method handles (send a-foo :win :something-else):

```
(defmethod (foo :win) (suboperation)
  (list 'something-random suboperation))
```

:case methods are unusual in that one flavor can have many :case methods for the same operation, as long as they are for different suboperations.

The suboperations :which-operations, :operation-handled-p, :send-if-handles and :get-handler-for are all handled automatically based on the collection of :case methods that are present.

Methods of type :or are also allowed. They are called just before the primary method, and if one of them returns a non-nil value, that is the value of the operation, and no more methods are called.

Here is a table of all the method types recognized by the standard styles of method combination.

(no type) If no type is given to defmethod, a primary method is created. This is the most common type of method.

:before
:after Used for the before-daemon and after-daemon methods used by :daemon method combination.

:default If there are no untyped methods among any of the flavors being combined, then the :default methods (if any) are treated as if they were untyped. If there are any untyped methods, the :default methods are ignored.

Typically a base-flavor (see page 431) defines some default methods for certain of the operations understood by its family. When using the default kind of method combination these default methods are suppressed if another component provides a primary method.

:or
:and Used for :daemon-with-or and :daemon-with-and method combination. The :or methods are wrapped in an or, or the :and methods are wrapped in an and, together with the primary method, between the :before and :after methods.

:override Allows the features of :or method combination to be used together with daemons. If you specify :daemon-with-override method combination, you may use :override methods. The :override methods are executed first, until one of them returns non-nil. If this happens, that method's value(s) are returned and no more methods are used. If all the :override methods return nil, the :before, primary and :after methods are executed as usual.

In typical usages of this feature, the :override method usually returns nil and does nothing, but in exceptional circumstances it takes over the handling of the operation.

:or, :and, :progn, :list, :inverse-list, pass-on, :append, :nconc.
Each of these methods types is allowed in the method combination style of the same name. In those method combination styles, these typed methods work just like untyped ones, but all the typed methods are called before all the untyped ones.

:case :case methods are used by :case method combination.

These method types can be used with any method combination style; they have standard meanings independent of the method combination style being used.

:around An :around method is able to control when, whether and how the remaining methods are executed. It is given a continuation that is a function that will execute the remaining methods, and has complete responsibility for calling it or not, and deciding what arguments to give it. For the simplest behavior, the arguments should be the operation name and operation arguments that the :around method itself received; but sometimes the whole purpose of the :around method is to modify the arguments before the remaining methods see them.

The :around method receives three special arguments before the arguments of the operation itself: the *continuation*, the *mapping-table*, and the *original-argument-list*. The last is a list of the operation name and operation arguments. The simplest way for the :around method to invoke the remaining methods is to do

```
(lexpr-funcall-with-mapping-table
  continuation mapping-table
  original-argument-list)
```

In general, the *continuation* should be called with either funcall-with-mapping-table or lexpr-funcall-with-mapping-table, providing the *continuation*, the *mapping-table*, and the operation name (which you know because it is the same as in the defmethod), followed by whatever arguments the remaining methods are supposed to see.

```
(defflavor foo-one-bigger-mixin () ())

(defmethod (foo-one-bigger-mixin :around :set-foo)
           (cont mt ignore new-foo)
  (funcall-with-mapping-table cont mt :set-foo
                                       (1+ new-foo)))
```

is a mixin which modifies the :set-foo operation so that the value actually used in it is one greater than the value specified in the message.

:inverse-around

:inverse-around methods work like :around methods, but they are invoked at a different time and in a different order.

With :around methods, those of earlier flavor components components are invoked first, starting with the instantiated flavor itself, and those of earlier components are invoked within them. :inverse-around methods are invoked in the opposite order: si:vanilla-flavor would come first. Also, all :around methods and wrappers are invoked inside all the :inverse-around methods.

For example, the :inverse-around :init method for tv:sheet (a base flavor for all window flavors) is used to handle the init keywords :expose-p and :activate-p, which cannot be handled correctly until the window is entirely set up. They are handled in this method because it is guaranteed to be the first method invoked by the :init operation on any flavor of window (because no component of tv:sheet defines an :inverse-around method for this operation). All the rest of the work of making a new window valid takes place in this method's continuation; when the continuation returns, the window must be as valid as it will ever be, and it is ready to be exposed or activated.

:wrapper Used internally by **defwrapper.**

Note that if one flavor defines both a wrapper and an :around method for the same operation, the :around method is executed inside the wrapper.

:combined Used internally for automatically-generated *combined* methods.

The most common form of combination is :daemon. One thing may not be clear: when do you use a :before daemon and when do you use an :after daemon? In some cases the primary method performs a clearly-defined action and the choice is obvious: :before :launch-rocket puts in the fuel, and :after :launch-rocket turns on the radar tracking.

In other cases the choice can be less obvious. Consider the :init message, which is sent to a newly-created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the :before daemon of the instantiated flavor is called, then :before daemons of successively more basic flavors are called, and finally the :before daemon (if any) of the base flavor is called. Then the primary method is called. After that, the :after daemon for the base flavor is called, followed by the :after daemons at successively less basic flavors.

Now, if there is no interaction among all these methods, if their actions are completely independent, then it doesn't matter whether you use a :before daemon or an :after daemon. There is a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors communicate with each other. In the case of the :init operation, the *init-plist* can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.

This means that when a :before daemon has run, it must assume that none of the methods for this operation have run yet. But the :after daemon knows that the :before daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should "transmit" the information in a :before daemon, and the second one should "receive" it in an :after daemon. So while the :before daemons are run, information is "transmitted"; that is, instance variables get set up. Then, when the :after daemons are run, they can look at the instance variables and act on their values.

In the case of the :init method, the :before daemons typically set up instance variables of the object based on the init-plist, while the :after daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

The problems become most difficult when you are creating a network of instances of various flavors that are supposed to point to each other. For example, suppose you have flavors for "buffers" and "streams", and each buffer should be accompanied by a stream. If you create the stream in the :before :init method for buffers, you can inform the stream of its corresponding buffer with an init keyword, but the stream may try sending messages back to the buffer, which is not yet ready to be used. If you create the stream in the :after :init method for buffers, there will be no problem with stream creation, but some other :after :init methods of other mixins may have run and made the assumption that there is to be no stream. The only way to guarantee success is to create the stream in a :before method and inform it of its associated buffer by sending it a message from the buffer's :after :init method. This scheme—creating associated objects in :before methods but linking them up in :after methods—often avoids problems, because all the various associated objects used by various mixins at least exist when it is time to make other objects point to them.

Since flavors are not hierarchically organized, the notion of levels of abstraction is not rigidly applicable. However, it remains a useful way of thinking about systems.

21.12 Implementation of Flavors

An object that is an instance of a flavor is implemented using the data type dtp-instance. The representation is a structure whose first word, tagged with dtp-instance-header, points to a structure (known to the microcode as an "instance descriptor") containing the internal data for the flavor. The remaining words of the structure are value cells containing the values of the instance variables. The instance descriptor is a defstruct that appears on the si:flavor property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling operations, and information for accessing the instance variables.

defflavor creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the first argument being the operation. The microcode binds self to the object and binds those instance variables that are supposed to be special to the value cells in the instance. Then it passes on the operation and arguments to a funcallable hash table taken from the flavor-structure for this flavor.

When the funcallable hash table is called as a function, it hashes the first argument (the operation) to find a function to handle the operation and an array called a mapping table. The variable sys:self-mapping-table is bound to the mapping table, which tells the microcode how to access the lexical instance variables, those not defined to be special. Then the function is called. If there is only one method to be invoked, this function is that method; otherwise it is an automatically-generated function called the combined method (see page 413), which calls the appropriate methods in the right order. If there are wrappers, they are incorporated into this combined method.

The mapping table is an array whose elements correspond to the instance variables which can be accessed by the flavor to which the currently executing method belongs. Each element contains the position in self of that instance variable. This position varies with the other instance variables and component flavors of the flavor of self.

Each time the combined method calls another method, it sets up the mapping table required by that method—not in general the same one which the combined method itself uses. The mapping tables for the called methods are extracted from the array leader of the mapping table used by the combined method, which is kept in a local variable of the combined method's stack frame while sys:self-mapping-table is set to the mapping tables for the component methods.

sys:self-mapping-table *Variable*

Holds the current mapping table, which tells the running flavor method where in **self** to find each instance variable.

Ordered instance variables are referred to directly without going through the mapping table. This is a little faster, and reduces the amount of space needed for mapping tables. It is also the reason why compiled code contains the positions of the ordered instance variables and must be recompiled when they change.

21.12.1 Order of Definition

There is a certain amount of freedom to the order in which you do **defflavor**'s, **defmethod**'s, and **defwrapper**'s. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need 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 a method can be defined (with **defmethod** or **defwrapper**) its flavor must have been defined (with **defflavor**). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with **defflavor**) it is not necessary that all of its component flavors be defined already. This is to allow **defflavor**'s to be spread between files according to the modularity of a program, and to provide for mutually-dependent flavors. Methods can be defined for a flavor some of whose component flavors are not yet defined; however, in certain cases compiling those methods may produce a warning that an instance variable was declared special (because the system did not realize it was an instance variable). If this happens, you should fix the problem and recompile.

The methods automatically generated by the **:gettable-instance-variables** and **:settable-instance-variables** **defflavor** options (see page 424) are generated at the time the **defflavor** is done.

The first time a flavor is instantiated, or when **compile-flavor-methods** is done, the system looks through all of the component flavors and gathers various information. At this point an error is signaled if not all of the components have been **defflavor**'ed. This is also the time at which certain other errors are detected, for instance lack of a required instance-variable (see the **:required-instance-variables** **defflavor** option, page 425). The combined methods (see page 413) are generated at this time also, unless they already exist.

After a flavor has been instantiated, it is possible to make changes to it. Such changes affect all existing instances if possible. This is described more fully immediately below.

21.12.2 Changing a Flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another **defflavor** with the same name. You can add or modify methods by doing **defmethod**'s. If you do a **defmethod** with the same flavor-name, operation (and suboperation if any), and (optional) method-type as an existing method, that method is replaced by the new definition. You can remove a method with **undefmethod** (see page 419).

These changes always propagate to all flavors that depend upon the changed flavor. Normally the system propagates the changes to all existing instances of the changed flavor and its dependent flavors. However, this is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message, on the ***error-output*** stream, to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they only work for the new version of the flavor, then trying to use those methods with the old instances won't work.

21.13 Useful Editor Commands

This section briefly documents some editor commands that are useful in conjunction with flavors.

Meta-.

The **Meta-.** (**Edit Definition**) command can find the definition of a flavor in the same way that it can find the definition of a function.

Edit Definition can find the definition of a method if you give it a suitable function spec starting with **:method**, such as **(:method tv:sheet :expose)**. The keyword **:method** may be omitted if the definition is in the editor already. Completion is available on the flavor name and operation name, as usual only for definitions loaded into the editor.

Meta-X Describe Flavor

Asks for a flavor name in the mini-buffer and describes its characteristics. When typing the flavor name you have completion over the names of all defined flavors (thus this command can be used to aid in guessing the name of a flavor). The display produced is mouse sensitive where there are names of flavors and of methods; as usual the right-hand mouse button gives you a menu of editor commands to apply to the name and the left-hand mouse button does one of them, typically positioning the editor to the source code for that name.

Meta-X List Methods
Meta-X Edit Methods

Asks you for an operation in the mini-buffer and lists all the flavors that have a method for that operation. You may type in the operation name, point to it with the mouse, or let it default to the operation of the message being sent by the Lisp form the cursor is on. List Methods produces a mouse-sensitive display allowing you to edit selected methods or just to see which flavors have methods, while Edit Methods skips the display and proceeds directly to editing the methods.

As usual with this type of command, the editor command Control-Shift-P advances the editor cursor to the next method in the list, reading in its source file if necessary. Typing Control-Shift-P, while the display is on the screen, edits the first method.

In addition, you can find a copy of the list in the editor buffer *Possibilities*. While in that buffer, the command Control-/ visits the definition of the method described on the line the cursor is pointing at.

These techniques of moving through the objects listed apply to all the following commands as well.

Meta-X List Combined Methods
Meta-X Edit Combined Methods

Asks you for an operation name and a flavor in two mini-buffers and lists all the methods that would be called to handle that operation for an instance of that flavor.

List Combined Methods can be very useful for telling what a flavor will do in response to a message. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them; type Control-Shift-P to get to successive ones.

Meta-X List Flavor Components
Meta-X Edit Flavor Components

Asks you for a flavor and lists or begins visiting all the flavors it depends on.

Meta-X List Flavor Dependents
Meta-X Edit Flavor Dependents

Asks you for a flavor and lists or begins visiting all the flavors that depend on it.

Meta-X List Flavor Direct Dependents
Meta-X Edit Flavor Direct Dependents

Asks you for a flavor and lists or begins visiting all the flavors that depend directly on it.

Meta-X List Flavor Methods
Meta-X Edit Flavor Methods

Asks you for a flavor and lists or begins visiting all the methods defined for that flavor. (This does not include methods inherited from its component flavors.)

21.14 Property List Operations

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. For more details and examples, see the general discussion of property lists (section 5.10, page 113). The usual property list functions (**get**, **putprop**, etc.) all work on instances by sending the instance the corresponding message.

si:property-list-mixin *Flavor*

This mixin flavor provides the basic operations on property lists.

:get *property-name* &optional *default* *Operation on* **si:property-list-mixin**

Looks up the object's *property-name* property. If it finds such a property, it returns the value; otherwise it returns *default*.

:getl *property-name-list* *Operation on* **si:property-list-mixin**

Like the **:get** operation, except that the argument is a list of property names. The **:getl** operation searches down the property list until it finds a property whose property name is one of the elements of *property-name-list*. It returns the portion of the property list begining with the first such property that it found. If it doesn't find any, it returns **nil**.

:putprop *value property-name* *Operation on* **si:property-list-mixin**

Gives the object an *property-name* property of *value*.

```
(send object :set :get property-name value)
```

also has this effect.

:remprop *property-name* *Operation on* **si:property-list-mixin**

Removes the object's *property-name* property, by splicing it out of the property list. It returns one of the cells spliced out, whose car is the former value of the property that was just removed. If there was no such property to begin with, the value is **nil**.

:get-location-or-nil *property-name* *Operation on* **si:property-list-mixin**
:get-location *property-name* *Operation on* **si:property-list-mixin**

Both return a locative pointer to the cell in which this object's *property-name* property is stored. If there is no such property, **:get-location-or-nil** returns **nil**, but **:get-location** adds a cell to the property list and initialized to **nil**, and a pointer to that cell is returned.

:push-property *value property-name* *Operation on* **si:property-list-mixin**

The *property-name* property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This operation sets the *property-name* property of the object to a list whose car is *value* and whose cdr is the former *property-name* property of the list. This is analogous to doing

```
(push value (get object property-name))
```

See the **push** special form (page 88).

`:property-list` *Operation on* si:property-list-mixin
Returns the list of alternating property names and values that implements the property list.

`:property-list-location` *Operation on* si:property-list-mixin
Returns a locative pointer to the cell in the instance which holds the property list data.

`:set-property-list` *list* *Operation on* si:property-list-mixin
Sets the list of alternating property names and values that implements the property list to *list*. So does

```
(send object :set :property-list list)
```

`:property-list` *list* *Init option for* si:property-list-mixin
This initializes the list of alternating property names and values that implements the property list to *list*.

21.15 Printing Flavor Instances Readably

A flavor instance can print out so that it can be read back in, as long as you give it a :print-self method that produces a suitable printed representation, and provide a way to parse it. The convention for doing this is to print as

#⊂flavor-name additional-data⊃

and make sure that the flavor defines or inherits a :read-instance method that can parse the *additional-data* and return an instance (see page 527). A convenient way of doing this is to use si:print-readably-mixin.

`si:print-readably-mixin` *Flavor*
Provides for flavor instances to print out using the #⊂ syntax, and also for reading things that were printed in that way.

`:reconstruction-init-plist` *Operation on* si:print-readably-mixin
When you use si:print-readably-mixin, you must define the operation :reconstruction-init-plist. This should return an alternating list of init options and values that could be passed to make-instance to create an instance "like" this one. Sufficient similarity is defined by the practical purposes of the flavor's implementor.

21.16 Copying Instances

Many people have asked "How do I copy an instance?" and have expressed surprise when told that the flavor system does not include any built-in way to copy instances. Why isn't there just a function copy-instance that creates a new instance of the same flavor with all its instance variables having the same values as in the original instance? This would work for the simplest use of flavors, but it isn't good enough for most advanced uses of flavors. A number of issues are raised by copying:

* Do you or do you not send an :init message to the new instance? If you do, what init-plist options do you supply?
* If the instance has a property list, you should copy the property list (e.g. with copylist) so that putprop or remprop on one of the instances does not affect the properties of the other

instance.

* 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. Should the copy talk to the same agent, or should a new agent be constructed for it?

* 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?

In general, you can see that in order to copy an instance one must understand a lot about the instance. One must know what the instance variables mean so that the values of the instance variables can be copied if necessary. One must understand what relations to the external environment the instance has so that new relations can be established for the new instance. One must even understand what the general concept 'copy' means in the context of this particular instance, and whether it means anything at all.

Copying is a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to 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 send it a message, as in **(send object :copy)**. It is up to you to implement the operation in a suitable fashion, such as

```
(defflavor foo (a b c) ()
  (:inittable-instance-variables a b))

(defmethod (foo :copy) ()
  (make-instance 'foo :a a :b b))
```

The flavor system chooses not to provide any default method for copying an instance, and does not even suggest a standard name for the copying message, because copying involves so many semantic issues.

If a flavor supports the **:reconstruction-init-plist** operation, a suitable copy can be made by invoking this operation and passing the result to **make-instance** along with the flavor name. This is because the definition of what the **:reconstruction-init-plist** operation should do requires it to address all the problems listed above. Implementing this operation is up to you, and so is making sure that the flavor implements sufficient init keywords to transmit any information that is to be copied. See page 446.

22. The I/O System

Zetalisp provides a powerful and flexible system for performing input and output to peripheral devices. Device independent I/O is generalized in the concept of an *I/O stream*. A stream is a source or sink for data in the form of characters or integers; sources are called *input streams* and sinks are called *output streams*. A stream may be capable of use in either direction, in which case it is a *bidirectional* stream. In a few unusual cases, it is useful to have a 'stream' which supports neither input nor output; for example, opening a file with direction :probe returns one (page 583). Streams on which characters are transferred are called *character streams*, and are used more often than *binary streams*, which usually transfer integers of type (unsigned-byte *n*) for some *n*.

Streams automatically provide a modular separation between the program which implements the stream and the program which uses it, because streams obey a standard protocol. The stream protocol is a special case is based on the general message passing protocol: a stream operation is invoked by calling the stream as a function, with a first argument that is a keyword and identifies the I/O operation desired (such as, :tyi to read a character) and additional arguments as that operation calls for them. The stream protocol consists of a particular set of operation names and calling conventions for them. It is documented in section 22.3, page 459.

Many programs do not invoke the stream operations directly; instead, they call standard I/O functions which then invoke stream operations. This is done for two reasons: the functions may provide useful services, and they may be transportable to Common Lisp or Maclisp. Programs that use stream operations directly are not transportable outside Zetalisp. The I/O functions are documented in the first sections of this chapter.

The generality of the Zetalisp I/O stream comes from the fact that I/O operations on it can invoke arbitrary Lisp code. For example, it would be very simple to implement a "morse code" stream that accepted character output and used beep with appropriate pauses to 'display' it. How to implement a stream is documented in section 22.3.12, page 474, and the following sections.

The most commonly used streams are windows, which read input from the keyboard and dispose of output by drawing on the screen, file streams, editor buffer streams which get input from the text in a buffer and insert output into the buffer, and string streams which do likewise with the contents of a string.

Another unusual aspect of Lisp I/O is the ability to input and output general Lisp objects, represented as text. These are done using the read and related functions and using print and related functions. They are documented in chapter 23.

22.1 Input Functions

The input functions read characters, lines, or bytes from an input stream. This argument is called *stream*. If omitted or nil, the current value of *standard-input*. This is the "default input stream", which in simple use reads from the terminal keyboard. If the argument is t, the current value of *terminal-io* is used; this is conventionally supposed to access "the user's terminal" and nearly always reads from the keyboard in processes belonging to windows.

If the stream is an interactive one, such as the terminal, the input is echoed, and functions which read more than a single character allow editing as well. peek-char echoes all of the characters that were skipped over if read-char would have echoed them; the character not removed from the stream is not echoed either.

When an input stream has no more data to return, it reports end of file. Each stream input operation has a convention for how to do this. The input functions accept an argument *eof-option* or two arguments *eof-error* and *eof-value* to tell them what to do if end of file is encountered instead of any input. The functions that take two *eof-* arguments are the Common Lisp ones. For them, end of file is an error if *eof-error* is non-nil or if it is unsupplied. If *eof-error* is nil, then the function returns *eof-value* at end of file.

The functions which have one argument called *eof-option* are from Maclisp. End of file causes an error if the argument is not supplied. Otherwise, end of file causes the function to return the argument's value. 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*.

sys:end-of-file (error) *Condition*

All errors signaled to report end of file possess this condition name.

The :stream operation on the condition instance returns the stream on which end of file was reached.

22.1.1 String Input Functions

read-line &optional *stream* (*eof-errorp* t) *eof-value ignore options*

Reads a line of text, terminated by a Return. It returns the line as a character string, *without* the Return character that ended the line. The argument *ignore* must be accepted for the sake of the Common Lisp specifications but it is not used.

This function is usually used to get a line of input from the user. If rubout processing is happening, then *options* is passed as the list of options to the rubout handler (see section 22.5, page 500).

There is a second value, t if the line was terminated by end of file.

readline &optional *stream eof-option options*

Like read-line but uses the Maclisp convention for specifying what to do about end of file. This function can take its first two arguments in the other order, for Maclisp compatibility only; see the note in section 22.1.3, page 451.

readline-trim &optional *stream eof-option options*

This is like readline except that leading and trailing spaces and tabs are discarded from the value before it is returned.

readline-or-nil &optional *stream eof-option options*

Like readline-trim except that nil is returned if the line is empty or all blank.

read-delimited-string &optional *delimiter stream eof rubout-handler-options buffer-size*

Reads input from *stream* until a delimiter character is reached, then returns as a string all the input up to but not including the delimiter. *delimiter* is either a character or a list of characters which all serve as delimiters. It defaults to the character End. *stream* defaults to the value of *standard-input*.

If *eof* is non-nil, then end of file on attempting to read the first character is an error. Otherwise it just causes an empty string to be returned. End of file once at least one character has been read is never an error but it does cause the function to return all the input so far.

Input is done using rubout handling and echoing if stream supports the :rubout-handler operation. In this case, *rubout-handler-options* are passed as the options argument to that operation.

buffer-size specifies the size of string buffer to allocate initially.

The second value returned is t if input ended due to end of file.

The third value is the delimiter character which terminated input, or nil if input terminated due to end of file. This character is currently represented as a fixnum, but perhaps someday will be a character object instead.

22.1.2 Character-Level Input Functions

read-char &optional *stream* (*eof-errorp* t) *eof-value*

Reads a character from *stream* and returns it as a character object. End of file is an error if *eof-errorp* is non-nil; otherwise, it causes read-char to return *eof-value*. This uses the :tyi stream operation.

read-byte *stream* &optional (*eof-errorp* t) *eof-value*

Like read-char but returns an integer rather than a character object. In strict Common Lisp, only read-char can be used on character streams and only read-byte can be used on binary streams.

read-char-no-hang &optional *stream* (*eof-errorp* t) *eof-value*

Similar but returns nil immediately when no input is available on an interactive stream. Uses the :tyi-no-hang stream operation (page 466).

unread-char *char* &optional *stream*

Puts *char* back into *stream* so that it will be read again as the next input character. *char* must be the same character that was read from stream most recently. It may not work to unread two characters in a row before reading again. Uses the :untyi stream operation (page 461).

peek-char *peek-type* &optional *stream* (*eof-errorp* t) *eof-value*

If *peek-type* is nil, this is like read-char except leaves the character to be read again by the next input operation.

If *peek-type* is t, skips whitespace characters and peeks at the first nonwhitespace character. That character is the value, and is also left to be reread.

If *peek-type* is a character, reads input until that character is seen. That character is unread and also returned.

listen &optional *stream*

t if input is now available on *stream*. Uses the :listen operation (page 466).

clear-input &optional *stream*

Discards any input now available on *stream*, if it is an interactive stream. Uses the :clear-input stream operation (page 469).

22.1.3 Maclisp Compatibility Input Functions

These functions accept an argument *eof-option* to tell them what to do if end of file is encountered instead of any input. End of file signals an error if the argument is not supplied. Otherwise, end of file causes the function to return the argument's value. 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*.

The arguments *stream* and *eof-option* can also be given in the reverse order for compatibility with old Maclisp programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an si:io-stream-p property whose value is t.

tyi &optional *stream eof-option*

Reads one character from *stream* and returns it. The character is echoed if *stream* is interactive, except that **Rubout** is not echoed. The **Control**, **Meta**, etc. shifts echo as **C-**, **M-**, etc.

The :tyi stream operation is preferred over the tyi function for some purposes. Note that it does not echo. See page 461.

(This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

readch &optional *stream eof-option*

Like tyi except that instead of returning a fixnum character, it returns a symbol whose print name is the character. The symbol is interned in the current package. This is just Maclisp's version of character object. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

This function is provided only for Maclisp compatibility, since in Zetalisp never uses symbols to represent characters in this way.

tyipeek &optional *peek-type stream eof-option*

This function is provided mainly for Maclisp compatibility; the :tyipeek stream operation is usually clearer (see page 461).

What tyipeek does depends on the *peek-type*, which defaults to nil. With a *peek-type* of nil, tyipeek returns the next character to be read from *stream*, without actually removing it from the input stream. The next time input is done from *stream* the character will still be there; in general, (= (tyipeek) (tyi)) is t.

If *peek-type* is a fixnum less than 1000 octal, then tyipeek reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is t, then tyipeek skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of tyipeek supported by Maclisp in which *peek-type* is a fixnum not less than 1000 octal is not supported, since the readtable formats of the Maclisp reader and the Zetalisp reader are quite different.

Characters passed over by tyipeek are echoed if *stream* is interactive.

22.1.4 Interactive Input with Prompting

prompt-and-read *type-of-parsing format-string* &rest *format-args*

Reads some sort of object from *query-io*, parsing it according to *type-of-parsing*, and prompting by calling format using *format-string* and *format-args*.

type-of-parsing is either a keyword or a list starting with a keyword and continuing with a list of options and values, whose meanings depend on the keyword used.

Most keywords specify reading a line of input and parsing it in some way. The line can be terminated with Return or End. Sometimes typing just End has a special meaning.

The keywords defined are

:eval-sexp
:eval-form
This keyword directs prompt-and-read to accept a Lisp expression. It is evaluated, and the value is returned by prompt-and-read.

If the Lisp expression is not a constant or quoted, the user is asked to confirm the value it evaluated to.

A default value can be specified with an option, as in

```
(:eval-sexp :default default)
```

Then, if the user types Space, prompt-and-read returns the *default* as the first value and :default as the second value.

:eval-sexp-or-end
:eval-form-or-end
Synonymously direct prompt-and-read to accept a Lisp expression or just the character End. If End is typed, prompt-and-read returns nil as its first value and :end as its second value. Otherwise, things proceed as for :eval-sexp.

A default value is allowed, as in :eval-sexp.

:read
:expression
Synonymously direct prompt-and-read to read an object and return it, with no evaluation.

:expression-or-end
Is like :expression except that the user is also allowed to type just End. If he does so, prompt-and-read returns the two values nil and :end.

:number
Directs prompt-and-read to read and return a number. It insists on getting a number, forcing the user to rub out anything else. Additional features can be specified with options:

```
(:number :input-radix radix :or-nil nil-ok-flag)
```

parses the number using radix *radix* if the number is a rational. (By default, the ambient radix is used). If *nil-ok-flag* is non-nil, then the user is also permitted to type just Return or End, and then nil is returned.

:decimal-number
:number-or-nil
:decimal-number-or-nil
Abbreviations for
```
(:number :input-radix 10)
(:number :or-nil t)
(:number :input-radix 10 :or-nil t)
```

:date Directs prompt-and-read to read a date and time, terminated with *Return* or End, and return it as a universal time (see page 777). It allows several options:

(:date :never-p *never-ok* :past-p *past-required*)

If *past-required* is non-nil, the date must be before the present time, or the user must rub out and use a different date. If *never-ok* is non-nil, the user may also type "never"; then nil is returned.

:date-or-never
:past-date
:past-date-or-never
Abbreviations for
```
(:date :never-p t)
(:date :past-p t)
(:date :never-p t :past-p t)
```

:character Directs prompt-and-read to read a single character and return a character object representing it.

:string Directs prompt-and-read to read a line and return its contents as a string, using **readline**.

:string-or-nil Directs prompt-and-read to read a line and return its contents as a string, using readline-trim. In addition, if the result would be empty, **nil** is returned instead of the empty string.

:string-list Like :string-trim but regards the line as a sequence of input strings separated by commas. Each substring between commas is trimmed, and a list of the strings is returned.

:keyword-list Like :string-list but converts each string to a keyword by interning it in the keyword package. The value is therefore a list of keywords.

:font-list Like :string-list but converts each string to a font name by interning it in the fonts package. The symbols must already exist in that package or the user is required to retype the input.

:delimited-string
Directs prompt-and-read to read a string terminated by specified delimiters. With

(:delimited-string :delimiter *delimiter-list*
                   :buffer-size *size*)

you can specify a list of delimiter characters and an initial size for the buffer. The list defaults to (#\end) and the size to 100.

The work is done by **read-delimited-string** (page 450). The delimiters and size are passed to that function.

:delimited-string-or-nil
Like **:delimited-string** except that **nil** is returned instead of the empty string if the first character read is a delimiter.

:host Directs **prompt-and-read** to read a line and interpret the contents as a network host name. The value returned is the host, looked up using **si:parse-host** (page 576). An option is defined:

```
(:host :default default-name :chaos-only chaos-only)
```

If the line read is empty, the host named *default-name* is used. If *chaos-only* is non-nil, only hosts on the Chaosnet are permitted input.

:host-list Like **:host** but regards the line as a sequence of host names separated by commas. Each host name is looked up as in **:host** and a list of the resulting hosts is returned.

:pathname-host
Like **:host** but uses **fs:get-pathname-host** to look up the host object from its name (page 577). Thus, you find hosts that can appear in pathnames rather than hosts that are on the network.

:pathname Directs **prompt-and-read** to read a line and parse it as a pathname, merging it with the defaults. If the line is empty, the default pathname is used. These options are defined:

```
(:pathname :defaults defaults-alist-or-pathname
           :version default-version)
```

uses *defaults-alist-or-pathname* as the *defaults* argument to **fs:merge-pathname-defaults**, and *default-version* as the *version* argument to it.

:pathname-or-nil
Is like **:pathname**, but if the user types just **End** it is interpreted as meaning "no pathname" rather than "use the default". Then **nil** is returned.

:pathname-list
Like **:pathname** but regards the line as a sequence of filenames separated by commas. Each filename is parsed and defaulted and a list of the resulting pathnames is returned.

:fquery Directs **prompt-and-read** to query the user for a fixed set of alternatives, using **fquery**. *type-of-parsing* should always be a list, whose car is **:fquery** and whose cdr is a list to be passed as the list of options (**fquery**'s first argument).
Example:

```
(prompt-and-read '(:fquery
                   . ,format:y-or-p-options)
                 "Eat it? ")
```

is equivalent to

```
(y-or-n-p "Eat it? ")
```

This keyword is most useful as a way to get to fquery when going through an interface defined to call prompt-and-read.

22.2 Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of *standard-output*. If *stream* is nil, the value of *standard-output* (i.e. the default) is used. If it is t, the value of *terminal-io* is used (i.e. the interactive terminal). This is all more-or-less compatible with Maclisp, except that instead of the variable *standard-output* Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

For print and the other expression output functions, see section 23.4, page 527.

write-char *char* &optional *stream*
tyo *char* &optional *stream*

Outputs *char* to *stream* (using :tyo). *char* may be an integer or a character object; in the latter case, it is converted to an integer before the :tyo.

write-byte *number* &optional *stream*

Outputs number to stream using :tyo. In strict Common Lisp, output to binary streams can be done only with write-byte and output to character streams requires write-char. In fact, the two functions are identical on the Lisp Machine.

write-string *string* &optional *stream* &key (*start* 0) *end*

Outputs *string* (or the specified portion of it) to *stream*.

write-line *string* &optional *stream* &key (*start* 0) *end*

Outputs *string* (or the specified portion) to *stream*, followed by a Return character.

fresh-line &optional *stream*

Outputs a Return character to stream unless either

(1) nothing has been output to *stream* yet, or

(2) the last thing output was a Return character, or

(3) *stream* does not remember what previous output there has been.

This uses the :fresh-line stream operation. The value is t if a Return is output, nil if nothing is output.

force-output &optional *stream*

Causes *stream*'s buffered output, if any, to be transmitted immediately. This uses the :force-output stream operation.

finish-output &optional *stream*
: Causes *stream*'s buffered output, if any, to be transmitted immediately, and waits until that is finished. This uses the :finish stream operation.

clear-output &optional *stream*
: Discards any output buffered in *stream*. This uses the :clear-output stream operation.

terpri &optional *stream*
: Outputs a Return character to *stream*. It returns t for Maclisp compatibility. It is wise not to depend on the value terpri returns.

cli:terpri &optional *stream*
: Outputs a Return character to *stream*. Returns nil to meet Common Lisp specifications. It is wise not to depend on the value cli:terpri returns.

The format function (see page 483) is very useful for producing nicely formatted text. It can do anything any of the above functions can do, and it makes it easy to produce good looking messages and such. format can generate a string or output to a stream.

stream-copy-until-eof *from-stream to-stream* &optional *leader-size*
: stream-copy-until-eof inputs characters from *from-stream* and outputs them to *to-stream*, until it reaches the end of file on the *from-stream*. For example, if x is bound to a stream for a file opened for input, then (stream-copy-until-eof x *terminal-io*) prints the file on the console.

  If *from-stream* supports the :line-in operation and *to-stream* supports the :line-out operation, then stream-copy-until-eof uses those operations instead of :tyi and :tyo, for greater efficiency. *leader-size* is passed as the argument to the :line-in operation.

beep &optional *beep-type* (*stream* *terminal-io*)
: This function is intended to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the :beep operation, then this function sends it a :beep message, passing *beep-type* along as an argument. Otherwise it just causes an audible beep on the terminal.

  beep-type is a keyword which explains the significance of this beep. Users can redefine beep to make different noises depending on the beep type. The defined beep types are:

  zwei:converse-problem
  : Used for the beep that is done when Converse is unable to send a message.

  zwei:converse-message-received
  : Used for the beeps done when a Converse message is received.

  zwei:no-completion
  : Used when you ask for completion in the editor and the string does not complete.

  tv:notify
  : Used for the beep done when you get a notification that cannot be printed on the selected window.

fquery	Used for the beep done by yes-or-no-p or by fquery with the :beep option specified.
supdup:terminal-bell	Used for the beep requested by the remote host being used through a Supdup window.
nil	Used whenever no other beep type applies.

The :beep operation is described on page 467.

cursorpos &rest *args*

This function exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages (see the Window System manual).

cursorpos normally operates on the *standard-output* stream; however, if the last argument is a stream or t (meaning *terminal-io*) then cursorpos uses that stream and ignores it when doing the operations described below. cursorpos only works on streams that are capable of these operations, such as windows. A stream is taken to be any argument that is not a number and not a symbol, or that is a symbol other than nil with a name more than one character long.

(cursorpos) => (*line* . *column*), the current cursor position.

(cursorpos *line column*) moves the cursor to that position. It returns t if it succeeds and nil if it doesn't.

(cursorpos *op*) performs a special operation coded by *op*, and returns t if it succeeds and nil if it doesn't. *op* is tested by string comparison, it is not a keyword symbol and may be in any package.

f	Moves one space to the right.
b	Moves one space to the left.
d	Moves one line down.
u	Moves one line up.
t	Homes up (moves to the top left corner). Note that t as the last argument to cursorpos is interpreted as a stream, so a stream *must* be specified if the t operation is used.
z	Home down (moves to the bottom left corner).
a	Advances to a fresh line. See the :fresh-line stream operation.
c	Clears the window.
e	Clear from the cursor to the end of the window.
l	Clear from the cursor to the end of the line.
k	Clear the character position at the cursor.
x	b then k.

22.3 I/O Streams

An *I/O stream*, or just *stream*, is a source and/or sink of characters or bytes. A set of *operations* is available with every stream; operations include things like "output a character" and "input a character". The way to perform an operation on a stream is the same for all streams, although what happens inside the stream is very different depending on what kind of a stream it is. So all a program has to know is how to deal with streams using the standard, generic operations. A programmer creating a new kind of stream only needs to implement the appropriate standard operations.

A stream is a message-receiving object. This means that it is something that you can apply to arguments. The first argument is a keyword symbol which is the name of the operation you wish to perform. The rest of the arguments depend on what operation you are doing. Message-passing and generic operations are explained in the flavor chapter (chapter 21, page 401).

Some streams can only do input, some can only do output, and some can do both. Some operations are only supported by some streams. Also, there are some operations that the stream may not support by itself, but which work anyway, albeit slowly, because the *stream default handler* can handle them. All streams support the operation :which-operations, which returns a list of the names of all of the operations that are supported "natively" by the stream. (:which-operations itself is not in the list.)

All input streams support all the standard input operations, and all output streams support all the standard output operations. All bidirectional streams support both.

streamp *object*

According to Common Lisp, this returns t if *object* is a stream. In the Lisp machine, a stream is any object which can be called as a function with certain calling conventions. It is theoretically impossible to test for this. However, streamp does return t for any of the usual types of streams, and nil for any Common Lisp datum which is not a stream.

22.3.1 Standard Streams

There are several variables whose values are streams used by many functions in the Lisp system. These variables and their uses are listed here. By convention, variables that are expected to hold a stream capable of input have names ending with -input, and similarly for output. Those expected to hold a bidirectional stream have names ending with -io. The names with asterisks are synonyms introduced for the sake of Common Lisp.

standard-input *Variable*
standard-input *Variable*

In the normal Lisp top-level loop, input is read from *standard-input* (that is, whatever stream is the value of *standard-input*). Many input functions, including tyi and read, take a stream argument that defaults to *standard-input*.

standard-output *Variable*
standard-output *Variable*

In the normal Lisp top-level loop, output is sent to *standard-output* (that is, whatever stream is the value of *standard-output*). Many output functions, including tyo and print, take a stream argument that defaults to *standard-output*.

error-output *Variable*
error-output *Variable*

The value of *error-output* is a stream on which noninteractive error or warning messages should be printed. Normally this is the same as *standard-output*, but *standard-output* might be bound to a file and *error-output* left going to the terminal.

debug-io *Variable*
debug-io *Variable*

The value of *debug-io* is used for all input and output by the error handler. Normally this is a synonym for *terminal-io*. The value may be nil, which is regarded as equivalent to a synonym for *terminal-io*. This feature is provided because users often set *debug-io* by hand, and it is much easier to set it back to nil afterward than to figure out the proper synonym stream pointing to *terminal-io*.

query-io *Variable*
query-io *Variable*

The value of *query-io* is a stream that should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program may be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. *query-io* is used by fquery and related functions; see page 769.

terminal-io *Variable*
terminal-io *Variable*

The value of *terminal-io* is the stream that the program should use to talk to the user's console. In an interactive program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the screen. However, in a background process that has no window, *terminal-io* defaults to a stream that does not ever expect to be used. If it is used, perhaps by an error printout, it turns into a background window and requests the user's attention.

trace-output *Variable*
trace-output *Variable*

The value of *trace-output* is the stream on which the trace function prints its output.

standard-input, *standard-output*, *error-output*, *debug-io*, *trace-output*, and *query-io* are initially bound to synonym streams that pass all operations on to the stream that is the value of *terminal-io*. Thus any operations performed on those streams go to the keyboard and screen.

Most user programs should not change the value of *terminal-io*. A program which wants (for example) to divert output to a file should do so by binding the value of *standard-output*; that way queries on *query-io*, debugging on *debug-io* and error messages sent to *error-output* can still get to the user by going through *terminal-io*, which is usually what is desired.

22.3.2 Standard Input Stream Operations

:tyi &optional *eof* *Operation on streams*

The stream inputs one character and returns it. For example, if the next character to be read in by the stream is a 'C', then the form

```
    (send s :tyi)
```

returns the value of #/C (that is, 103 octal). Note that the :tyi operation does not echo the character in any fashion; it just does the input. The tyi function (see page 452) does echoing when reading from the terminal.

The optional *eof* argument to the :tyi operation tells the stream what to do if it gets to the end of the file. If the argument is not provided or is nil, the stream returns nil at the end of file. Otherwise it signals a sys:end-of-file error. Note that this is *not* the same as the eof-option argument to read, tyi, and related functions.

The :tyi operation on a binary input stream returns a non-negative number, not necessarily to be interpreted as a character.

For some streams (such as windows), not all the input data are numbers. Some are lists, called *blips*. The :tyi operation returns only numbers. If the next available input is not a number, it is discarded, and so on until a number is reached (or end of file is reached).

:any-tyi &optional *eof* *Operation on streams*

Like :tyi but returns any kind of datum. Non-numbers are not discarded as they would be by :tyi. This distinction only makes a difference on streams which can provide input which is not composed of numbers; currently, only windows can do that.

:tyipeek &optional *eof* *Operation on streams*

Peeks at the next character or byte from the stream without discarding it. The next :tyi or :tyipeek operation will get the same character.

eof is the same as in the :tyi operation: if nil, end of file returns nil; otherwise, it signals a sys:end-of-file error.

:untyi *char* *Operation on streams*

Unreads the character or byte *char*; that is to say, puts it back into the input stream so that the next :tyi operation will read it again. For example,

```
    (send s :untyi 120)
    (send s :tyi) ==> 120
```

This operation is used by read, and any stream that supports :tyi must support :untyi as well.

You are only allowed to :untyi one character before doing a :tyi, and the character you :untyi must be the last character read from the stream. That is, :untyi can only be used to back up one character, not to stuff arbitrary data into the stream. You also can't :untyi after you have peeked ahead with :tyipeek since that does one :untyi itself. Some streams implement :untyi by saving the character, while others implement it by backing up the pointer to a buffer.

:string-in *eof-option string* &optional (*start* 0) *end* *Operation on streams*

Reads characters from the stream and stores them into the array *string*. Many streams can implement this far more efficiently that repeated :tyi's. *start* and *end*, if supplied, delimit the portion of *string* to be stored into. If *eof-option* is non-nil then a sys:end-of-file error is signaled if end of file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end of file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

Two values are returned: the index of the next position in *string* to be filled, and a flag that is non-nil if end of file was reached before *string* was filled. Most callers do not need to look at either of these values.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

:line-in &optional *leader* *Operation on streams*

The stream should input one line from the input source, and return it as a string with the carriage return character stripped off. Contrary to what you might assume from its name, this operation is not much like the readline function.

Many streams have a string that is used as a buffer for lines. If this string itself were returned, there would be problems caused if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. In order to solve this problem, the string must be copied. On the other hand, some streams don't reuse the string, and it would be wasteful to copy it on every :line-in operation. This problem is solved by using the *leader* argument to :line-in. If *leader* is nil (the default), the stream does not bother to copy the string and the caller should not rely on the contents of that string after the next operation on the stream. If *leader* is t, the stream does make a copy. If *leader* is a fixnum then the stream makes a copy with an array leader *leader* elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches end of file while reading in characters, it returns the characters it has read in as a string and returns a second value of t. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was a whole line or just the trailing characters after the last carriage return in the input source.

This operation should be implemented by all input streams whose data are characters.

:string-line-in *eof-option string* &optional (*start* 0) *end* *Operation on streams*

Reads characters, storing them in *string*, until *string* is full or a Return character is read. If input stops due to a Return, the Return itself is not put in the buffer.

Thus, this operation is nearly the same as :string-in, except that :string-in always keeps going until the buffer is full or until end of file.

start and *end*, if supplied, delimit the portion of *string* to be stored into. If *eof-option* is non-nil then a sys:end-of-file error is signaled if end of file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end of file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

Three values are returned:

(1) The index in *string* at which input stopped. This is the first index not stored in.

(2) t if input stopped due to end of file.

(3) t if the line is incomplete; that is, if a Return character did not terminate it.

:read-until-eof *Operation on streams*

Discards all data from the stream until it is at end of file, or does anything else with the same result.

:close &optional *ignore* *Operation on streams*

Releases resources associated with the stream, when it is not going to be used any more. On some kinds of streams, this may do nothing. On Chaosnet streams, it closes the Chaosnet connection, and on file streams, it closes the input file on the file server.

The argument is accepted for compatibility with :close on output streams.

22.3.3 Standard Output Stream Operations

:tyo *char* *Operation on streams*

The stream outputs the character *char*. For example, if s is bound to a stream, then the form

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

outputs a B to the stream. For binary output streams, the argument is a non-negative number rather than specifically a character.

:fresh-line *Operation on streams*

Tells the stream that it should position itself at the beginning of a new line. If the stream is already at the beginning of a fresh line it should do nothing; otherwise it should output a carriage return. If the stream cannot tell whether it is at the beginning of a line, it should always output a carriage return.

:string-out (*string* 0) &optional *start end* *Operation on streams*

Outputs the characters of *string* successively to *stream*. This operation is provided for two reasons; first, it saves the writing of a loop which is used very often, and second, many streams can perform this operation much more efficiently than the equivalent sequence of :tyo operations.

If *start* and *end* are not supplied, the whole string is output. Otherwise a substring is output; *start* is the index of the first character to be output (defaulting to 0), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle :string-out must check for them and interpret them appropriately.

:line-out *string* &optional (*start* 0) *end* *Operation on streams*

Outputs the characters of *string* successively to *stream*, then outputs a Return character. *start* and *end* optionally specify a substring, as with :string-out. If the stream doesn't support :line-out itself, the default handler implements it by means of :tyo.

This operation should be implemented by all output streams whose data are characters.

:close &optional *mode* *Operation on streams*

Closes the stream to make the output final if this is necessary. The stream becomes *closed* and no further output operations should be performed on it. However, it is all right to :close a closed stream. On many file server hosts, a file being written is not accessible to be read until the output stream is closed.

This operation does nothing on streams for which it is not meaningful.

The *mode* argument is normally not supplied. If it is :abort, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly-created file is deleted; it will be as if it was never opened in the first place. Any previously existing file with the same name remains undisturbed.

:eof *Operation on streams*

Indicates the end of data on an output stream. This is different from :close because some devices allow multiple data files to be transmitted without closing. :close implies :eof when the stream is an output stream and the close mode is not :abort.

This operation does nothing on streams for which it is not meaningful.

22.3.4 Asking Streams What They Can Do

All streams are supposed to support certain operations which enable a program using the stream to ask which operations are available.

:which-operations *Operation on streams*

Returns a list of operations handled natively by the stream. Certain operations not in the list may work anyway, but slowly, so it is just as well if any programs that work with or without them choose not to use them.

:which-operations itself need not be in the list.

:operation-handled-p *operation* *Operation on streams*

Returns t if *operation* is handled natively by the stream: if *operation* is a member of the :which-operations list, or is :which-operations.

:send-if-handles *operation* &rest *arguments* *Operation on streams*

Performs the operation *operation*, with the specified *arguments*, only if the stream can handle it. If *operation* is handled, this is the same as sending an *operation* message directly, but if *operation* is not handled, using :send-if-handles avoids any error.

If *operation* is handled, :send-if-handles returns whatever values the execution of the *operation* returns. If *operation* is not handled, :send-if-handles returns nil.

:direction *Operation on streams*

Returns :input, :output, or :bidirectional for a bidirectional stream.

There are a few kinds of streams, which cannot do either input or output, for which the :direction operation returns nil. For example, open with the :direction keyword specified as nil returns a stream-like object which cannot do input or output but can handle certain file inquiry operations such as :truename and :creation-date.

:characters *Operation on streams*

Returns t if the data input or output on the stream represent characters, or nil if they are just numbers (as for a stream reading a non-text file).

:element-type *Operation on streams*

Returns a type specified describing in principle the data input or output on the stream. Refer to the function stream-element-type, below, which works using this operation.

These functions for inquiring about streams are defined by Common Lisp.

input-stream-p *stream*
t if *stream* handles input operations (at least, if it handles :tyi).

output-stream-p *stream*
t if *stream* handles output operations (at least, if it handles :tyo).

stream-element-type *stream*
Returns a type specifier which describes, conceptually, the kind of data input from or output to *stream*. The value is always a subtype of integer (for a binary stream) or a subtype of character (for a character stream). If it is a subtype of integer, a Common Lisp program should use read-byte (page 450) or write-byte (page 456) for I/O. If it is a subtype of character, read-char (page 450) or write-char (page 456) should be used.

The value returned is not intended to be rigidly accurate. It describes the typical or characteristic sort of data transferred by the stream, but the stream may on occasion deal with data that do not fit the type; also, not all objects of the type may be possible as input or even make sense as output. For example, windows describe their element type as character even though they may offer blips, which are lists, as input on occasion. In addition, streams which say they provide characters really return integers if the :tyi operation is used rather than the standard Common Lisp function read-char.

22.3.5 Operations for Interactive Streams

The operations :listen, :tyi-no-hang, :rubout-handler and :beep are intended for interactive streams, which communicate with the user. :listen and :tyi-no-hang are supported in a trivial fashion by other streams, for compatibility.

:listen *Operation on streams*
On an interactive device, the :listen operation returns non-nil if there are any input characters immediately available, or nil if there is no immediately available input. On a non-interactive device, the operation always returns non-nil except at end of file.

The main purpose of :listen is to test whether the user has hit a key, perhaps trying to stop a program in progress.

:tyi-no-hang &optional *eof* *Operation on streams*
Just like :tyi except that it returns nil rather than waiting if it would be necessary to wait in order to get the character. This lets the caller check efficiently for input being available and get the input if there is any.

:tyi-no-hang is different from :listen because it reads a character.

Streams for which the question of whether input is available is not meaningful treat this operation just like :tyi. So do Chaosnet file streams. Although in fact reading a character from a file stream may involve a delay, these delays are *supposed* to be insignificant, so we pretend they do not exist.

:any-tyi-no-hang &optional *eof* *Operation on streams*

Like :tyi-no-hang but does not filter and discard input which is not numbers. It is therefore possible to see blips in the input stream. The distinction matters only for input from windows.

:rubout-handler *options function* &rest *args* *Operation on streams*

This is supported by interactive bidirectional streams, such as windows on the terminal, and is described in its own section below (see section 22.5, page 500).

:beep &optional *type* *Operation on streams*

This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. *beep-type* is a keyword selecting among several different beeping noises; see beep (page 457) for a list of them.

22.3.6 Cursor Positioning Stream Operations

:read-cursorpos &optional (*units* :pixel) *Operation on streams*

This operation is supported by all windows and some other streams.

It returns two values, the current *x* and *y* coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units *x* and *y* should be; the symbols :pixel and :character are understood. :pixel means that the coordinates are measured in display pixels (bits), while :character means that the coordinates are measured in characters horizontally and lines vertically.

This operation and :increment-cursorpos are used by the format ~T request (see page 487), which is why ~T doesn't work on all streams. Any stream that supports this operation should support :increment-cursorpos as well.

Some streams return a meaningful value for the horizontal position but always return zero for the vertical position. This is sufficient for ~T to work.

:increment-cursorpos *Operation on streams*

x-increment y-increment &optional (*units* :pixel)

Moves the stream's cursor left or down according to the specified increments, as if by outputting an appropriate number of space or return characters. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

Any stream which supports this operation should support :read-cursorpos as well, but it need not support :set-cursorpos.

Moving the cursor with :increment-cursorpos differs from moving it to the same place with :set-cursorpos in that this operation is thought of as doing output and :set-cursorpos is not. For example, moving a window's cursor down with :increment-cursorpos when it is near the bottom to begin with will wrap around, possibly doing a **MORE**. :set-cursorpos, by comparison, cannot move the cursor "down" if it is at the bottom of the window; it can move the cursor explicitly to the top of the window, but then no **MORE** will happen.

Some streams, such as those created by with-output-to-string, cannot implement arbitrary cursor motion, but do implement this operation.

:set-cursorpos *x y* &optional (*units* :pixel) *Operation on streams*

This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

:clear-screen *Operation on streams*

Erases the screen area on which this stream displays. Non-window streams don't support this operation.

There are many other special-purpose stream operations for graphics. They are not documented here, but in the window-system documentation. No claim that the above operations are the most useful subset should be implied.

22.3.7 Operations for Efficient Pretty-Printing

grindef runs much more efficiently on streams that implement the :untyo-mark and :untyo operations.

:untyo-mark *Operation on streams*

This is used by the grinder (see page 528) if the output stream supports it. It takes no arguments. The stream should return some object that indicates how far output has gotten up to in the stream.

:untyo *mark* *Operation on streams*

This is used by the grinder (see page 528) in conjunction with :untyo-mark. It takes one argument, which is something returned by the :untyo-mark operation of the stream. The stream should back up output to the point at which the object was returned.

22.3.8 Random Access File Operations

The following operations are implemented only by streams to random-access devices, principally files.

:read-pointer *Operation on streams*

Returns the current position within the file, in characters (bytes in fixnum mode). For text files on ASCII file servers, this is the number of Lisp Machine characters, not ASCII characters. The numbers are different because of character-set translation.

:set-pointer *new-pointer* *Operation on streams*

Sets the reading position within the file to *new-pointer* (bytes in fixnum mode). For text files on ASCII file servers, this does not do anything reasonable unless *new-pointer* is 0, because of character-set translation. Some file systems support this operation for input streams only.

:rewind *Operation on streams*
This operation is obsolete. It is the same as :set-pointer with argument zero.

22.3.9 Buffered Stream Operations

:clear-input *Operation on streams*
Discards any buffered input the stream may have. It does nothing on streams for which it is not meaningful.

:clear-output *Operation on streams*
Discards any buffered output the stream may have. It does nothing on streams for which it is not meaningful.

:force-output *Operation on streams*
This is for output streams to buffered asynchronous devices, such as the Chaosnet. :force-output causes any buffered output to be sent to the device. It does not wait for it to complete; use :finish for that. If a stream supports :force-output, then :tyo, :string-out, and :line-out may have no visible effect until a :force-output is done.

This operation does nothing on streams for which it is not meaningful.

:finish *Operation on streams*
This is for output streams to buffered asynchronous devices, such as the Chaosnet. :finish does a :force-output, then waits until the currently pending I/O operation has been completed.

This operation does nothing on streams for which it is not meaningful.

The following operations are implemented only by buffered input streams. They allow increased efficiency by making the stream's internal buffer available to the user.

:read-input-buffer &optional *eof* *Operation on streams*
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. These values are similar to the *string*, *start*, *end* arguments taken by many functions and stream operations. If the end of the file has been reached and no input bytes are available, this operation returns nil or signals an error, based on the *eof* argument, just like the :tyi operation. After reading as many bytes from the array as you care to, you must use the :advance-input-buffer operation.

:get-input-buffer &optional *eof* *Operation on streams*
This is an obsolete operation similar to :read-input-buffer. The only difference is that the third value is the number of significant elements in the buffer-array, rather than a final index. If found in programs, it should be replaced with :read-input-buffer.

:advance-input-buffer &optional *new-pointer* *Operation on streams*

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.

22.3.10 Obtaining Streams to Use

Windows are one important class of streams. Each window can be used as a stream. Output is displayed on the window and input comes from the keyboard. A window is created using make-instance on a window flavor. Simple programs use windows implicitly through *terminal-io* and the other standard stream variables.

Also important are *file streams*, which are produced by the function open (see page 582). These read or write the contents of a file.

Chaosnet streams are made from Chaosnet connections. Data output to the stream goes out over the network; data coming in over the network is available as input from the stream. File streams that deal with Chaosnet file servers are very similar to Chaosnet streams, but Chaosnet streams can be used for many purposes other than file access.

String streams read or write the contents of a string. They are made by with-output-to-string or with-input-from-string (see page 473), or by make-string-input-stream or make-string-output-stream, below.

Editor buffer streams read or write the contents of an editor buffer.

The *null stream* may be passed to a program that asks for a stream as an argument. It returns immediate end of file if used for input and throws away any output. The null stream is the symbol si:null-stream. This is to say, you do not call that function to get a stream or use the symbol's value as the stream; *the symbol itself* is the object that is the stream.

The *cold-load stream* is able to do I/O to the keyboard and screen without using the window system. It is what is used by the error handler, if you type Terminal Call, to handle a background error that the window system cannot deal with. It is called the cold-load stream because it is what is used during system bootstrapping, before the window system has been loaded.

si:null-stream *operation* &rest *arguments*

This function is the null stream. Like any stream, it supports various operations. Output operations are ignored and input operations report end of file immediately, with no data. Usage example:

```
(let ((*standard-output* 'si:null-stream))
  (function-whose-output-I-dont-want))
```

si:cold-load-stream *Constant*

The one and only cold-load stream. Usage example:

```
(let ((*query-io* si:cold-load-stream))
  (yes-or-no-p "Clear all window system locks? "))
```

with-open-stream (*variable expression*) *body...* *Macro*

body is executed with *variable* bound to the value of *expression*, which ought to be a stream. On exit, whether normal or by throwing, a :close message with argument :abort is sent to the stream.

This is a generalization of with-open-file, which is equivalent to using with-open-stream with a call to open as the *expression*.

with-open-stream-case (*variable expression*) *clauses...* *Macro*

Like with-open-stream as far as opening and closing the stream are concerned, but instead of a simple body, it has clauses like those of a condition-case that say what to do if *expression* does or does not get an error. See with-open-file-case, page 580.

make-synonym-stream *symbol-or-locative*
make-syn-stream *symbol-or-locative*

Creates and returns a *synonym* stream ('syn' for short). Any operations sent to this stream are redirected to the stream that is the value of the argument (if it is a symbol) or the contents of it (if it is a locative).

A synonym stream is actually an uninterned symbol whose function defnition is forwarded to the function cell of the argument or to the contents of the argument as appropriate. If the argument is a symbol, the synonym stream's print-name is *symbol*-syn-stream; otherwise the name is just syn-stream. Once a synonym stream is made for a symbol, it is recorded, and the same one is handed out again if there is another request for it.

The two names for this function are synonyms too.

make-concatenated-stream &rest *streams*

Returns an input stream which will read its input from the first of *streams* until that reaches its eof, then read input from the second of *streams*, and so on until the last of *streams* has reached end of file.

make-two-way-stream *input-stream output-stream*

Returns a bidirectional stream which passes input operations to *input-stream* and passes output operations to *output-stream*. This works by attempting to recognize all standard input operations; anything not recognized is passed to *output-stream*.

make-echo-stream *input-stream output-stream*

Like make-two-way-stream except that each input character read via *input-stream* is output to *output-stream* before it is returned to the caller.

make-broadcast-stream &rest *streams*

Returns a stream that only works in the output direction. Any output sent to this stream is forwarded to all of the streams given. The :which-operations is the intersection of the :which-operations of all of the streams. The value(s) returned by a stream operation are the values returned by the last stream in *streams*.

zwei:interval-stream *interval-or-from-bp* &optional *to-bp in-order-p hack-fonts*

Returns a bidirectional stream that reads or writes all or part of an editor buffer. Note that editor buffer streams can also be obtained from **open** by using a pathname whose host is ED, ED-BUFFER or ED-FILE (see section 24.7.6, page 575).

The first three arguments specify the buffer or portion to be read or written. Either the first argument is an *interval* (a buffer is one kind of interval), and all the text of that interval is read or written, or the first two arguments are two buffer pointers delimiting the range to be read or written. The third argument is used only in the latter case; if non-nil, it tells the function to assume that the second buffer pointer comes later in the buffer than the first and not to take the time to verify the assumption.

The stream has only one pointer inside it, used for both input and output. As you do input, the pointer advances through the text. When you do output, it is inserted in the buffer at the place where the pointer has reached. The pointer starts at the beginning of the specified range.

hack-fonts tells what to do about fonts. Its possible values are

t	The character ε is recognized as special when you output to the stream; sequences such as ε2 are interpreted as font-changes. They do not get inserted into the buffer; instead, they change the font in which following output will be inserted. On input, font change sequences are included to indicate faithfully what was in the buffer.
:tyo	You are expected to read and write 16-bit characters containing font numbers.
nil	All output is inserted in font zero and font information is discarded in the input you receive. This is the best mode to use if you are reading or otherwise parsing the contents of an editor buffer.

sys:with-help-stream (*stream options...*) *body...* *Macro*

Executes the *body* with the variable *stream* bound to a suitable stream for printing a large help message. If *standard-output* is a window, then *stream* is also a window; a temporary window which fills the screen. Otherwise, *stream* is just the same as *standard-output*.

The purpose of this is to spare the user the need to read a large help printout in a small window, or have his data overwritten by it permanently. This is the mechanism used if you type the Control-Help key while in the rubout handler.

options is a list of alternating keywords and values.

:label	The value (which is evaluated) is used as the label of the temporary window, if one is used.
:width	The value, which is not evaluated, is a symbol. While *body* is executed, this symbol is bound to the width, in characters, available for the message.

:height	The value is a symbol, like the value after :width, and it is bound to the height in lines of the area available for the help message.
:superior	The value, which is evaluated, specifies the original stream to use in deciding where to print the help message. The default is ***standard-output***.

22.3.11 String I/O Streams

The functions and special forms in this section allow you to create I/O streams that input from or output to the contents of a string.

make-string-input-stream *string* &optional (*start* 0) *end*

Returns a stream which can be used to read the contents of *string* (or the portion of it from index *start* to index *end*) as input. End of file occurs on reading past position *end* or the end of string.

make-string-output-stream &optional *string*

Returns an output stream which will accumulate all output in a string. If *string* is non-nil, output is added to it with string-nconc (page 216). Otherwise, a new string is created and used to hold the output.

get-output-stream-string *string-output-stream*

Returns the string of output accumulated so far by a stream which was made by make-string-output-stream. The accumulated output is cleared out, so it will not be obtained again if get-output-stream-string is called another time on the same stream.

with-input-from-string (*var string* &key *start end index*) *body*... *Macro*

The form

```
(with-input-from-string (var string)
   body)
```

evaluates the forms in *body* with the variable *var* bound to a stream which reads characters from the string which is the value of the form *string*. The value of the construct is the value of the last form in its body.

If the *start* and *end* arguments are specified, they should be forms. They are evaluated at run time to produce the indices starting and ending the portion of *string* to be read.

If the *index* argument is specified, it should be something setf can store in. When *body* is finished, the index in the string at which reading stopped is stored there. This is the index of the first character not read. If the entire string was read, it is the length of the string. The value of *index* is not updated until with-input-from-string is exited, so you can't use its value within the body to see how far the reading has gotten. Example:

```
(with-input-from-string
     (foo "This is a test." :start (+ 2 2) :end 8 :index bar)
   (readline))
```

returns " is " and sets bar to eight.

An older calling sequence which used positional rather than keyword arguments is still accepted:

(with-input-from-string (*var string index end*)
  body)

The functions **read-from-string** and **cli:read-from-string** are convenient special cases of what **with-input-from-string** can do. See page 533.

with-output-to-string (*var* [*string* [*index*]]) *body*... *Macro*

This special form provides a variety of ways to send output to a string through an I/O stream.

(with-output-to-string (*var*)
  body)

evaluates the forms in *body* with *var* bound to a stream which saves the characters output to it in a string. The value of the special form is the string.

(with-output-to-string (*var string*)
  body)

appends its output to the string which is the value of the form *string*. (This is like the **string-nconc** function; see page 216.) The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have a fill pointer. If *string* is too small to contain all the output, **adjust-array-size** is used to make it bigger.

(with-output-to-string (*var string index*)
  body)

is similar to the above except that *index* is a variable or **setf**-able reference which contains the index of the next character to be stored into. It must be initialized before the **with-output-to-string** and it is updated upon normal exit. The value of *index* is not updated until **with-output-to-string** returns, so you can't use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if there is one, it is updated on exit.

Another way of doing output to a string is to use the **format** facility (see page 483).

22.3.12 Implementing Streams

There are two ways to implement a stream: using **defun** or using **flavors**.

Using flavors is best when you can take advantage of the predefined stream mixins, including those which perform buffering, or when you wish to define several similar kinds of streams that can inherit methods from each other.

defun (or **defselect**, which is a minor variation of the technique) may have an advantage if you are dividing operations into broad groups and handling them by passing them off to one or more other streams. In this case, the automatic operation decoding provided by flavors may get in the way. A number of streams in the system are implemented using **defun** or **defselect** for historical reasons. It isn't yet clear whether there is any reason not to convert most of them to use flavors.

If you use **defun**, you can use the *stream default handler* to implement some of the standard operations for you in a default manner. If you use flavors, there are predefined mixins to do this for you.

A few streams are individual objects, one of a kind. For example, there is only one null stream, and no need for more, since two null streams would behave identically. But most streams are elements of a general class. For example, there can be many file streams for different files, even though all behave the same way. There can also be multiple streams reading from different points in the same file.

If you implement a class of streams with **defun**, then the actual streams must be closures of the function you define, made with **closure**.

If you use flavors to implement the streams, having a class of similar streams comes naturally: each instance of the flavor is a stream, and the instance variables distinguish one stream of the class from another.

22.3.13 Implementing Streams with Flavors

To define a stream using flavors, define a flavor which incorporates the appropriate predefined stream flavor, and then redefine those operations which are peculiar to your own type of stream.

Flavors for defining unbuffered streams:

si:stream *Flavor*

This flavor provides default definitions for a few standard operations such as **:direction** and **:characters**. Usually you do not have to mention this explicitly; instead you use the higher level flavors below, which are built on this one.

si:input-stream *Flavor*

This flavor provides default definitions of all the mandatory input operations except **:tyi** and **:untyi**, in terms of those two. You can make a simple non-character input stream by defining a flavor incorporating this one and giving it methods for **:tyi** and **:untyi**.

si:output-stream *Flavor*

This flavor provides default definitions of all the mandatory output operations except **:tyo**, in terms of **:tyo**. All you need to do to define a simple unbuffered non-character output stream is to define a flavor incorporating this one and give it a method for the **:tyo** operation.

si:bidirectional-stream *Flavor*

This is a combination of **si:input-stream** and **si:output-stream**. It defines **:direction** to return **:bidirectional**. To define a simple unbuffered non-character bidirectional stream, build on this flavor and define **:tyi**, **:untyi** and **:tyo**.

The unbuffered streams implement operations such as :string-out and :string-in by repeated use of :tyo or :tyi.

For greater efficiency, if the stream's data is available in blocks, it is better to define a buffered stream. You start with the predefined buffered stream flavors, which define :tyi or :tyo themselves and manage the buffers for you. You must provide other operations that the system uses to obtain the next input buffer or to write or discard an output buffer.

Flavors for defining buffered streams:

si:buffered-input-stream *Flavor*

This flavor is the basis for a non-character buffered input stream. It defines :tyi as well as all the other standard input operations, but you must define the two operations :next-input-buffer and :discard-input-buffer, which the buffer management routines use.

:next-input-buffer *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations, such as :tyi, to get the next bufferful of input data. It should return three values: an array containing the data, a starting index in the array, and an ending index. For example, in a Chaosnet stream, this operation would get the next packet of input data and return pointers delimiting the actual data in the packet.

:discard-input-buffer *buffer-array* *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations such as :tyi. It says that the buffer management routines have used or thrown away all the input in a buffer, and the buffer is no longer needed.

In a Chaosnet stream, this operation would return the packet buffer to the pool of free packets.

si:buffered-output-stream *Flavor*

This flavor is the basis for a non-character buffered output stream. It defines :tyo as well as all the other standard output operations, but you must define the operations :new-output-buffer, :send-output-buffer and :discard-output-buffer, which the buffer management routines use.

:new-output-buffer *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to get an empty buffer for storing more output data. How the buffer is obtained depends on the kind of stream, but in any case this operation should return an array (the buffer), a starting index, and an ending index. The two indices delimit the part of the array that is to be used as a buffer.

For example, a Chaosnet stream would get a packet from the free pool and return indices delimiting the part of the packet array which can hold data bytes.

:send-output-buffer *buffer-array ending-index* — *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to send the data in a buffer that has been completely or partially filled.

ending-index is the first index in the buffer that has not actually been stored. This may not be the same as the ending index that was returned by the :new-output-buffer operation that was used to obtain this buffer; if a :force-output is being handled, *ending-index* indicates how much of the buffer is currently full.

The method for this operation should process the buffer's data and, if necessary, return the buffer to a free pool.

:discard-output-buffer *buffer-array* — *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :clear-output, to free an output buffer and say that the data in it should be ignored.

It should simply return *buffer-array* to a free pool, if appropriate.

Some buffered output streams simply have one buffer array which they use over and over. For such streams, :new-output-buffer can simply return that particular array each time; :send-output-buffer and :discard-output-buffer do not have to do anything about returning the buffer to a free pool. In fact, :discard-output-buffer can probably do nothing.

si:buffered-stream — *Flavor*

This is a combination of si:buffered-input-stream and si:buffered-output-stream, used to make a buffered bidirectional stream. The input and output buffering are completely independent of each other. You must define all five of the low level operations: :new-output-buffer, :send-output-buffer and :discard-output-buffer for output, and :next-input-buffer and :discard-input-buffer for input.

The data in most streams are characters. Character streams should support either :line-in or :line-out in addition to the other standard operations.

si:unbuffered-line-input-stream — *Flavor*

This flavor is the basis for unbuffered character input streams. You need only define :tyi and :untyi.

si:line-output-stream-mixin — *Flavor*

To make an unbuffered character output stream, mix this flavor into the one you define, together with si:output-stream. In addition, you must define :tyo, as for unbuffered non-character streams.

si:buffered-input-character-stream *Flavor*
This is used just like si:buffered-input-stream, but it also provides the :line-in operation and makes :characters return t.

si:buffered-output-character-stream *Flavor*
This is used just like si:buffered-output-stream, but it also provides the :line-out operation and makes :characters return t.

si:buffered-character-stream *Flavor*
This is used just like si:buffered-stream, but it also provides the :line-in and :line-out operations and makes :characters return t.

To make an unbuffered random-access stream, you need only define the :read-pointer and :set-pointer operations as appropriate. Since you provide the :tyi or :tyo handler yourself, the system cannot help you.

In a buffered random-access stream, the random access operations must interact with the buffer management. The system provides for this.

si:input-pointer-remembering-mixin *Flavor*
Incorporate this into a buffered input stream to support random access. This flavor defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide a definition for the :set-buffer-pointer operation. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *new-pointer* *Operation on* si:input-pointer-remembering-mixin
You must define this operation if you use si:input-pointer-remembering-mixin and want the :set-pointer operation to work.

This operation should arrange for the next :next-input-buffer operation to provide a bufferful of data that includes the specified character or byte position somewhere inside it.

The value returned should be the file pointer corresponding to the first character or byte of that next bufferful.

si:output-pointer-remembering-mixin *Flavor*
Incorporate this into a buffered output stream to support random access. This mixin defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide definitions for the :set-buffer-pointer and :get-old-data operations. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *Operation on* si:output-pointer-remembering-mixin
new-pointer
This is the same as in si:input-pointer-remembering-mixin.

:get-old-data *Operation on* si:output-pointer-remembering-mixin
buffer-array lower-output-limit

The buffer management routines perform this operation when you do a :set-pointer that is outside the range of pointers that fit in the current output buffer. They first send the old buffer, then do :set-buffer-pointer as described above to say where in the file the next output buffer should come, then do :new-output-buffer to get the new buffer. Then the :get-old-data operation is performed.

It should fill current buffer (*buffer-array*) with the *old* contents of the file at the corresponding addresses, so that when the buffer is eventually written, any bytes skipped over by random access will retain their old values.

The instance variable si:stream-output-lower-limit is the starting index in the buffer of the part that is supposed to be used for output. si:stream-output-limit is the ending index. The instance variable si:output-pointer-base is the file pointer corresponding to the starting index in the buffer.

si:file-stream-mixin *Flavor*

Incorporate this mixin together with si:stream to make a *file probe stream*, which cannot do input or output but records the answers to an enquiry about a file. You should specify the init option :pathname when you instantiate the flavor.

You must provide definitions for the :plist and :truename operations; in terms of them, this mixin defines the operations :get, :creation-date, and :info.

si:input-file-stream-mixin *Flavor*

Incorporate this mixin into input streams that are used to read files. You should specify the file's pathname with the :pathname init option when you instantiate the flavor.

In addition to the services and requirements of si:file-stream-mixin, this mixin takes care of mentioning the file in the who-line. It also includes si:input-pointer-remembering-mixin so that the :read-pointer operation, at least, will be available.

si:output-file-stream-mixin *Flavor*

This is the analogue of si:input-file-stream-mixin for output streams.

22.3.14 Implementing Streams Without Flavors

You do not need to use flavors to implement a stream. Any object that can be used as a function, and decodes its first argument appropriately as an operation name, can serve as a stream. Although in practice using flavors is as easy as any other way, it is educational to see how to define streams "from scratch".

We could begin to define a simple output stream, which accepts characters and conses them onto a list, as follows:

```
(defvar the-list nil)

(defun list-output-stream (op &optional arg1 &rest rest)
  (ecase op
    (:tyo
     (setq the-list (cons arg1 the-list)))
    (:which-operations '(:tyo))))
```

This is an output stream, and so it supports the :tyo operation. All streams must support :which-operations.

The lambda-list for a stream defined with a **defun** must always have one required parameter (*op*), one optional parameter (*arg1*), and a rest parameter (*rest*).

This definition is not satisfactory, however. It handles :tyo properly, but it does not handle :string-out, :direction, :send-if-handles, and other standard operations.

The function **stream-default-handler** exists to spare us the trouble of defining all those operations from scratch in simple streams like this. By adding one additional clause, we let the default handler take care of all other operations, if it can.

```
(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 #'list-output-stream
                                  op arg1 rest))))
```

If the operation is not one that the stream understands (e.g. :string-out), it calls **stream-default-handler**. Note how the rest argument is passed to it. This is why the argument list must look the way it does. **stream-default-handler** can be thought of as a restricted analogue of flavor inheritance.

If we want to have only one stream of this sort, the symbol **list-output-stream** can be used as the stream. The data output to it will appear in the global value of **the-list**. One more step is required, though:

```
(defprop list-output-stream t si:io-stream-p)
```

This tells certain functions including **read** to treat the symbol **list-output-stream** as a stream rather than as an end of file option.

If we wish to be able to create any number of list output streams, each accumulating its own list, we must use closures:

```
(defvar the-stream nil
  "Inside a list output stream, holds the stream itself.")
(defvar the-list nil
  "Inside a list output stream,
holds the list of characters being accumulated.")

(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (push arg1 the-list)))
        (:withdrawal (prog1 the-list (setq the-list nil)))
        (:which-operations '(:tyo :withdrawal))
        (otherwise
          (stream-default-handler the-stream
                                  op arg1 rest))))

(defun make-list-output-stream ()
  (let ((the-stream the-list))
    (setq the-stream
          (closure '(the-stream the-list)
                   'list-output-stream))))
```

We have added a new operation :withdrawal that can be used to find out what data has been accumulated by a stream. This is necessary because we can no longer simply look at or set the global value of the-list; that is not the same as the value closed into the stream.

In addition, we have a new variable the-stream which allows the function list-output-stream to know which stream it is serving at any time. This variable is passed to stream-default-handler so that when it simulates :string-out by means of :tyo, it can do the :tyo's to the same stream that the :string-out was done to.

The same stream could be defined with defselect instead of defun. It actually makes only a small difference. The defun for list-output-stream could be replaced with this code:

```
(defselect (list-output-stream list-output-d-h)
  (:tyo (arg1)
    (push arg1 the-list))
  (:withdrawal ()
    (prog1 the-list (setq the-list nil))))

(defun list-output-d-h (op &optional arg1 &rest rest)
  (stream-default-handler the-stream op arg1 rest))
```

defselect takes care of decoding the operations, provides a definition for :which-operations, and allows you to write a separate lambda list for each operation.

By comparison, the same stream defined using flavors looks like this:

```
(defflavor list-output-stream ((the-list nil))
           (si:line-output-stream-mixin si:output-stream))

(defmethod (list-output-stream :tyo) (character)
  (push character the-list))

(defmethod (list-outut-stream :withdrawal) ()
  (prog1 the-list (setq the-list nil)))

(defun make-list-output-stream ()
  (make-instance 'list-output-stream))
```

Here is a simple input stream, which generates successive characters of a list.

```
(defvar the-list)        ;Put your input list here
(defvar the-stream)
(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 (pop the-list))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler the-stream
                              op arg1 rest))))

(defun make-list-input-stream (the-list)
  (let (the-stream untyied-char)
    (setq the-stream
          (closure '(the-list the-stream untyied-char)
                   'list-input-stream))))
```

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.

stream-default-handler *stream op arg1 rest*

Tries to handle the *op* operation on *stream*, given arguments of *arg1* and the elements of *rest*. The exact action taken for each of the defined operations is explained with the documentation on that operation, above.

22.4 Formatted Output

There are two ways of doing general formatted output. One is the function **format**. The other is the **output** subsystem. **format** uses a control string written in a special format specifier language to control the output format. **format:output** provides Lisp functions to do output in particular formats.

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 obscure and hard to read. Then **format:output** becomes advantageous because it works with ordinary Lisp control constructs.

22.4.1 The Format Function

format *destination control-string* &rest *args*

Produces formatted output. **format** outputs the characters of *control-string*, except that a tilde ('~') introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. 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.

The output is sent to *destination*. If *destination* is **nil**, a string is created which contains the output; this string is returned as the value of the call to **format**. In all other cases **format** returns no interesting value (generally it returns **nil**). If *destination* is a stream, the output is sent to it. If *destination* is **t**, the output is sent to ***standard-output***. If *destination* is a string with an array-leader, such as would be acceptable to **string-nconc** (see page 216), the output is added to the end of that string.

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (':') and atsign ('@') modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"            ; This is an S directive with no parameters.
"~3,4:@s"       ; This is an S directive with two parameters, 3 and 4,
                ;   and both the colon and atsign flags.
"~,4S"          ; The first prefix parameter is omitted and takes
                ;   on its default value, while the second is 4.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use **format** efficiently. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (" ' ") followed by the desired character may be used as a prefix parameter, so that you don't have to know the decimal numeric values of characters in the character set. For example, you can use "~5,'0d" instead of "~5,48d" to print a decimal number in five columns with leading zeros.

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 doesn't really 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.

Here are some relatively simple examples to give you the general flavor of how **format** is used.

```
(format nil "foo") => "foo"
(setq x 5)
(format nil "The answer is ~D." x) => "The answer is 5."
(format nil "The answer is ~3D." x) => "The answer is   5."
(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
(format nil "The character ~:@C is strange." #\meta-beta)
     => "The character Meta-β (Greek-b) is strange."
(setq n 3)
(format nil "~D item~:P found." n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
     => "three dogs are here."
(format nil "~R dog~:*~[~1; is~:;s are~] here." n)
     => "three dogs are here."
(format nil "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
     => "Here are three puppies."
```

The directives will now be described. *arg* will be used to refer to the next argument from *args*.

~A *arg*, any Lisp object, is printed without escaping (as by **princ**). ~:A prints ( ) if *arg* is nil; this is useful when printing something that is always supposed to be a list. ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol,colinc,minpad,padchar*A is the full form of ~A, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~S This is just like ~A, but *arg* is printed *with* escaping (as by **prin1** rather than **princ**).

~D *arg*, a number, is printed in base ten. Unlike print, ~D never puts a decimal point after the number. ~*n*D uses a column width of *n*; spaces are inserted on the left if the number requires less than *n* columns for its digits and sign. If the number doesn't fit in *n* columns, additional columns are used as needed. ~*n*,*m*D uses *m* as the pad character instead of space. If *arg* is not a number, it is printed in ~A format and decimal base. The @ modifier causes the number's sign to be printed always; the default is only to print it if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

~O This is just like ~D but prints in octal instead of decimal.

~X This is just like ~D but prints in hex instead of decimal. Note that ~X used to have a different meaning: print one or more spaces. Uses of ~X intended to have this meaning should be replaced with ~@T.

~B This is just like ~D but prints in binary instead of decimal.

~*w*,*d*,*k*,*ovfl*,*pad* F

arg is printed in nonexponential floating point format, as in '10.5'. (If the magnitude of *arg* is very large or very small, it is printed in exponential notation.) The parameters control the details of the formatting.

w is the total field width desired. If omitted, this is not constrained.

d is the number of digits to print after the decimal point. If *d* is omitted, it is chosen to do a good job based on *w* (if specified) and the value of *arg*.

k is a scale factor. *arg* is multiplied by (exp 10. *k*) before it is printed.

ovfl is a character to use for overflow. If *arg* is too big to print and fit the constraints of field width, etc., and *ovfl* is specified then the whole field is filled with *ovfl*. If *ovfl* is not specified, *arg* is printed using extra width as needed.

pad is a character to use for padding on the left, when the field width is specified and not that many characters are really needed.

If the @ modifier is used, a sign is printed even if *arg* is positive.

Rational numbers are converted to floats and then printed. Anything else is printed with ~*w*D format.

~*w*,*d*,*e*,*k*,*ovfl*,*pad*,*expt* E

arg is printed in exponential notation, as in '.105e+2'. The parameters control the details of the formatting.

w is the total field width desired. If omitted, this is not constrained.

d and *k*

control the number of mantissa digits and their arrangement around the decimal point. $d+1$ digits are printed. If k is positive, all of them are significant digits, and the decimal point is printed after the first k of them. If k is zero or negative, the first $|k|+1$ of the $d+1$ digits are leading zeros, and the decimal point follows the first zero. (This zero can be omitted if necessary to fit the number in *w* characters.) So the number of significant figures is less than *d* if *k* is negative.

The exponent printed always compensates for any powers of ten introduced according to *k*, so 10.5 might be printed as 0.105e+2 or as 1050.0e-2.

If *d* is omitted, the system chooses enough significant figures to represent the float accurately. If *k* is omitted, the default is one.

e is the number of digits to use for the exponent. If it is not specified, however many digits are needed are used.

ovfl is the overflow character. If the exponent doesn't fit in *e* digits or the entire number does not fit in *w* characters, then if *ovfl* is specified, the field of *w* characters is filled with *ovfl*. Otherwise more characters are used as needed.

pad is a character to use for padding on the left, when the field width is specified and not that many characters are really needed.

expt is a character to use to separate the mantissa from the exponent. The default is e or s or f, whichever would be used in printing the number normally.

If the @ modifier is used, a sign is printed even if *arg* is positive.

~*w,d,e,k,ovfl,pad,expt* G
Prints a floating point number *arg* in either ~F or ~E format. Fixed format is used if the absolute value of *arg* is less than (expt 10. *d*), and exponential format otherwise. (If *d* is not specified, it defaults based on the value of *arg*.) If fixed format is used, *e*+2 blanks are printed at the end (where the exponent and its separator and sign would go, in exponential format). These count against the width *w* if that is specified. Four blanks are used if *e* is omitted. The diminished width available, *d*, *ovfl* and *pad* are used as specified. The scale factor used in fixed format is always zero, not *k*.

If exponential format needs to be used, all the parameters are passed to the ~E directive to print the number.

Rational numbers are converted to floats and then printed. Anything else is printed with ~*w*D format.

~$ ~*rdig,ldig,field,padchar*$ prints *arg*, a float, with exactly *rdig* digits after the decimal point. The default for *rdig* is 2, which is convenient for printing amounts of money. At least *ldig* digits are printed preceding the decimal point; leading zeros are printed if there would be fewer than *ldig*. The default for *ldig* is 1. The number is right justified in a field *field* columns long, padded out with *padchar*. The colon modifier means that the sign character is to be at the beginning of the field, before the padding, rather than just to the left of the number. The atsign modifier says that the sign character should always be output.

If *arg* is not a number, or is unreasonably large, it is printed in ~*field,,,padchar*@A format; i.e. it is princ'ed right-justified in the specified field width.

~C (character *arg*) is put in the output. *arg* is treated as a keyboard character (see page 206), thus it may contain extra control-bits. These are printed first by representing them with abbreviated prefixes: 'C-' for Control, 'M-' for Meta, 'H-' for Hyper, and 'S-' for Super.

With the colon flag (~:C), the names of the control bits are spelled out (e.g. 'Control-Meta-F') and non-printing characters are represented by their names (e.g. 'Return') rather than being output as themselves. The printing characters Space and Altmode are also represented as their names, but all others are printed directly.

With both colon and atsign (~:@C), the colon-only format is printed, and then if the character requires the Top or Greek (Front) shift key(s) to type it, this fact is mentioned (e.g. '∀ (Top-U)'). This is the format used for telling the user about a key he is expected to type, for instance in prompt messages.

For all three of these formats, if the character is a mouse character, it is printed as Mouse-, the name of the button, '-', and the number of clicks.

With just an atsign (~@C), the character is printed in such a way that the Lisp reader can understand it, using '#\' or '#/', depending on the escaping character of *readtable* (see page 516).

~% Outputs a carriage return. ~*n*% outputs *n* carriage returns. No argument is used. Simply putting a carriage return in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

~& The :fresh-line operation is performed on the output stream. Unless the stream knows that it is already at the front of a line, this outputs a carriage return. ~*n*& does a :fresh-line operation and then outputs *n*-1 carriage returns.

~| Outputs a page separator character (#\page). ~*n*| does this *n* times. With a : modifier, 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. | is vertical bar, not capital I.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes.

~<CR> Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

~* *arg* is ignored. ~*n** ignores the next *n* arguments. ~:* "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. ~*n*:* backs up *n* arguments. ~*n*@* is absolute; it moves to argument *n* (*n* = 0 specifies the first argument).

When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~P If *arg* is not 1, a lower-case 's' is printed. ('P' is for 'plural'.) ~:P does the same thing, after doing a ~:*; that is, it prints a lower-case s if the *last* argument was not 1. ~@P prints 'y' if the argument is 1, or 'ies' if it is not. ~:@P does the same thing, but backs up first.

~T Spaces over to a given column. ~*n*,*m*T outputs sufficient spaces to move the cursor to column *n*. If the cursor is already past column *n*, it outputs spaces to move it to column $n+mk$, for the smallest integer value *k* possible. *n* and *m* default to 1. Without the colon flag, *n* and *m* are in units of characters; with it, they are in units of pixels.

Note: this operation works properly *only* on streams that support the :read-cursorpos and :increment-cursorpos stream operations (see page 467). On other streams, any ~T operation simply outputs two spaces. When format is creating a string, ~T works by assuming that the first character in the string is at the left margin.

~@T simply outputs a space. ~*rel* T simply outputs *rel* spaces. ~*rel*, *period* T outputs *rel* spaces and then additional spaces until it reaches a column which is a multiple of *period*. If the output stream does not support :read-cursorpos then it simply outputs *rel* spaces.

~R ~R prints *arg* as a cardinal English number, e.g. four. ~:R prints *arg* as an ordinal number, e.g. fourth. ~@R prints *arg* as a Roman numeral, e.g. IV. ~:@R prints *arg* as an old Roman numeral, e.g. IIII.

~*n*R prints *arg* in radix *n*. The flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

~? Uses up two arguments, and processes the first one as a format control string using the second one's elements as arguments. Thus,

```
        (format nil "~? ~D" "~O ~O" '(4 20.) 9)
```

returns "4 24 9".

~@? processes the following argument as a format control string, using all the remaining arguments. Any arguments it does not use are left to be processed by the format directives following the ~@? in the original control string.

```
        (format nil "~@? ~D" "~O ~O" 4 20. 9)
```

likewise returns "4 24 9".

~→*str*~← Performs the formatting specified by *str*, with indentation on any new lines. Each time a Return is printed during the processing of *str*, it is followed by indentation sufficient to line up underneath the place where the cursor was at the beginning of *str*. For example,

```
        (format t "Foo: ~8T~→~A~←" string)
```

prints *string* with each line starting at column 8. If *string* is (string-append "This is" #\return "the string") then the output is

```
        Foo:    This is
                the string
```

~(*str*~) Performs output with case conversion. The formatting specified by *str* is done, with all the letters in the resulting output being converted to upper or lower case according to the modifiers given to the ~(command:

~(without modifiers
Converts all the letters to lower case.

~:(Converts the first letter of each word to upper case and the rest to lower case.

~@(Converts the first letter of the first word to upper case, and all other letters to lower case.

~:@(Converts all the letters to upper case.

~1(Converts the first letter of the first word to upper case and does not change anything else. If you arrange to generate all output in lower case except for letters that should be upper case regardless of context, you can use this directive when the output appears at the beginning of a sentence.

Example:
```
    "~(FoO BaR~) ~:(FoO BaR~) ~@(FoO BaR~) ~:@(FoO BaR~)
    ~1(at the White Hart~)"
```
produces
```
    foo bar Foo Bar Foo bar FOO BAR
    At the White Hart
```

~[*str0*~;*str1*~;...~;*strn*~]

This is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example,
```
    "~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
       ~;Tiger ~;Yu-Shiang ~]kitty"
```
The *arg*th alternative is selected; 0 selects the first. If a prefix parameter is given (i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is '#'). If *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is instead ~:;, then the last clause is an "else" clause, which is performed if no other clause is selected. For example,
```
    "~[Siamese ~;Manx ~;Persian ~;Tiger ~
       ~;Yu-Shiang ~:;Bad ~] kitty"
```

~[~*tag00*,*tag01*,...;*str0*~*tag10*,*tag11*,...;*str1*...~] allows the clauses to have explicit tags. The parameters to each ~; are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If ~*a1*,*a2*,*b1*,*b2*,...:; (note the colon) is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. ~:; with no parameters may be used at the end to denote a default clause. For example,
```
    "~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z:;letter ~
       ~'0,'9:;digit ~:;other ~]"
```

~:[*false*~;*true*~] selects the *false* control string if *arg* is nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed. For example,
```
    (setq *print-level* nil *print-length* 5)
    (format nil
            "~@[ *PRINT-LEVEL*=~D~]~@[ *PRINT-LENGTH*=~D~]"
            prinlevel prinlength)
       =>  " *PRINT-LENGTH*=5"
```

The combination of ~[and # is useful, for example, for dealing with English conventions for printing lists:

```
(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."
```

~; Separates clauses in ~[and ~< constructions. It is undefined elsewhere.

~] Terminates a ~[. It is undefined elsewhere.

~{*str*~} This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to **format**. 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, then two elements of the list get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there can be at most *n* repetitions of processing of *str*. Here are some simple 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>."
```

Using ~^ as well, to terminate *str* if no arguments remain, we can print a list with commas between the elements:

```
(format nil "Elements: ~{~S~^, ~}." '(a b c))
     => "Elements: A, B, C."
```

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format nil "Pairs of things:~:{ <~S,~S>~}."
          '((a 1) (b 2) (c 3)))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs of things:~@{ <~S,~S>~}."
            'a 1 'b 2 'c 3)
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs of things:~:@{ <~S,~S>~}."
            '(a 1) '(b 2) '(c 3))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it does not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an 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 it must be processed at most once, and the ~:} says it must be processed at least once. Therefore it is processed exactly once, using **args** as the arguments.

As another example, the **format** function itself uses **format-error** (a routine internal to the **format** package) to signal error messages, which in turn uses **ferror**, which uses **format** recursively. Now **format-error** takes a string and arguments, just like **format**, but also prints some additional information: if the control string in **ctl-string** actually is a string (it might be a list—see below), then it prints the string and a little 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↓~%~3@T/"~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~]."
...
```

~} Terminates a ~{. It is undefined elsewhere.

~< *~mincol,colinc,minpad,padchar<text~>* justifies *text* within a field at least *mincol* wide. *text* may be divided up into segments with ~;—the spacing is evenly divided between the text segments. 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, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. *Minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. *mincol* defaults to 0. For example,

```
(format nil "~10<foo~;bar~>")              =>  "foo    bar"
(format nil "~10:<foo~;bar~>")             =>  "  foo  bar"
(format nil "~10:@<foo~;bar~>")            =>  "  foo bar "
(format nil "~10<foobar~>")                =>  "    foobar"
(format nil "~10:<foobar~>")               =>  "    foobar"
(format nil "~10@<foobar~>")               =>  "foobar    "
(format nil "~10:@<foobar~>")              =>  "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.5902)      =>  "$******2.59"
```

Note that *text* may include format directives. The last example illustrates how the ~< directive can be combined with the ~f directive to provide more advanced control over the formatting of numbers.

Here are some examples of the use of ~^ within a ~< construct. ~^ is explained in detail below, however the general idea is that it eliminates the segment in which it appears and all following segments if there are no more arguments.

```
(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"
```

The idea is that if a segment contains a ~^, and **format** runs out of arguments, it just stops there instead of getting an error, and it as well as the rest of the segments are ignored.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, then 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, then the text segment for the first clause is output before the padded text. The first clause ought to contain a carriage return (~%). The first clause is always processed, and so any arguments it refers to will be 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*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ';; '. The prefix parameter 1 in ~1:; accounts for the width of the comma which will follow 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, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
    "~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then format sees whether the stream handles the :size-in-characters message. If it does, then format sends that message and uses the first returned value as the line length in characters. If it doesn't, format uses 72. as the line length.

Rather than using this complicated syntax, one can often call the function format:print-list (see page 495).

~> Terminates a ~<. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. 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 ~< clause, because it aborts the entire clause. ~^ may appear anywhere in a ~{ construct.

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then 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 ~:^.

~Q An escape to arbitrary user-supplied code. *arg* is called as a function; its arguments are the prefix parameters to ~Q, if any. *args* can be passed to the function by using the V prefix parameter. The function may output to *standard-output* and may look at the variables format:colon-flag and format:atsign-flag, which are t or nil to reflect the : and @ modifiers on the ~Q. For example,

```
    (format t "~VQ" foo bar)
```

is a fancy way to say

```
    (funcall bar foo)
```

and discard the value. Note the reversal of order; the V is processed before the Q.

~\ This begins a directive whose name is longer than one character. The name is terminated by another \ character. The following directives have names longer than one character and make use of the ~\ mechanism as part of their operation.

~\lozenged-string\
: This is like ~A except when output is to a window, in which case the argument is printed in a small font inside a lozenge.

~\lozenged-character\
: This is like ~C except when output is to a window, in which case the argument is printed in a small font inside a lozenge if it has a character name, even if it is a formatting character or graphic character.

~\date\
: This expects an argument that is a universal time (see page 776), and prints it as a date and time using time:print-universal-date.
Example:

```
        (format t "It is now ~\date\" (get-universal-time))
```

prints

```
        It is now Saturday the fourth of December, 1982; 4:00:32 am
```

~\time\
: This expects an argument that is a universal time (see page 776), and prints it in a brief format using time:print-universal-time.
Example:

```
        (format t "It is now ~\time\" (get-universal-time))
```

prints

```
        It is now 12/04/82 04:01:38
```

~\datime\
: This prints the current time and date. It does not use an argument. It is equivalent to using the ~\time\ directive with (time:get-universal-time) as argument.

~\time-interval\
: This prints a time interval measured in seconds using the function time:print-interval-or-never.
Example:

```
        (format t "It took ~\time-interval\." 3601.)
```

prints

```
        It took 1 hour 1 second.
```

You can define your own directives. How to do this is not documented here; read the code. Names of user-defined directives longer than one character may be used if they are enclosed in backslashes (e.g. ~4,3\GRAPH\).

(Note: format also allows *control-string* to be a list. If the list is a list of one element, which is a string, the string is simply printed. This is for the use of the format:outfmt function below. The old feature wherein a more complex interpretation of this list was possible is now considered obsolete; use format:output if you like using lists.)

A condition instance can also be used as the *control-string*. Then the :report operation is used to print the condition instance; any other arguments are ignored. This way, you can pass a condition instance directly to any function that normally expects a format string and arguments.

format:print-list *destination element-format list* &optional *separator start-line tilde-brace-options*

This function provides a simpler interface for the specific purpose of printing comma-separated lists with no list element split across two lines; see the description of the ~:; directive (page 492) to see the more complex way to do this within **format**. *destination* tells where to send the output; it can be **t**, **nil**, a **string-nconc**'able string, or a stream, as with **format**. *element-format* is a **format** control-string that tells how to print each element of *list*; it is used as the body of a ~{...~} construct. *separator*, which defaults to ", " (comma, space) is a string which goes after each element except the last. **format** control commands are not recommended in *separator*. *start-line*, which defaults to three spaces, is a **format** control-string that is used as a prefix at the beginning of each line of output, except the first. **format** control commands are allowed in *separator*, but they should not swallow arguments from *list*. *tilde-brace-options* is a string inserted before the opening '{'; it defaults to the null string, but allows you to insert colon and/or atsign. The line-width of the stream is computed the same way that the ~:; command computes it; it is not possible to override the natural line-width of the stream.

22.4.2 The Output Subsystem

The formatting functions associated with the **format:output** subsystem allow you to do formatted output using Lisp-style control structure. Instead of a directive in a **format** control string, there is one formatting function for each kind of formatted output.

The calling conventions of most of the formatting functions are similar. The first argument is usually the datum to be output. The second argument is usually the minimum number of columns to use. The remaining arguments are keyword arguments.

Most of the functions accept the keyword arguments *padchar*, *minpad* and *tab-period*. *padchar* is a character to use for padding. *minpad* is a minimum number of padding characters to output after the data. *tab-period* is the distance between allowable places to stop padding. To make the meaning of *tab-period* clearer, if the value of *tab-period* is 5, if the minimum size of the field is 10, and if the value of *minpad* is 2, then a datum that takes 9 characters is padded out to 15 characters. The requirement to use at least two characters of padding means it can't fit into 10 characters, and the *tab-period* of 5 means the next allowable stopping place is at 10+5 characters. The default values for *minpad* and *tab-period*, if they are not specified, are zero and one. The default value for *padchar* is space.

The formatting functions always output to ***standard-output*** and do not require an argument to specify the stream. The macro **format:output** allows you to specify the stream or a string, just as **format** does, and also makes it convenient to concatenate constant and variable output.

format:output *stream string-or-form...* *Macro*

Makes it convenient to intersperse arbitrary output operations with printing of constant strings. ***standard-output*** is bound to *stream*, and each *string-or-form* is processed in succession from left to right. If it is a string, it is printed; otherwise it is a form, which is evaluated for effect. Presumably the forms will send output to ***standard-output***.

If *stream* is written as **nil**, then the output is put into a string which is returned by **format:output**. If *stream* is written as **t**, then the output goes to the prevailing value of ***standard-output***. Otherwise *stream* is a form, which must evaluate to a stream.

Here is an example:

```
(format:output t "FOO is " (prin1 foo) " now." (terpri))
```

Because **format:output** is a macro, what matters about *stream* is not whether it *evaluates* to **t** or **nil**, but whether it is actually written as **t** or **nil**.

format:outfmt *string-or-form...* *Macro*

Some system functions ask for a **format** control string and arguments, to be printed later. If you wish to generate the output using the formatted output functions, you can use **format:outfmt**, which produces a control argument that will eventually make **format** print the desired output (this is a list whose one element is a string containing the output). A call to **format:outfmt** can be used as the second argument to **ferror**, for example:

```
        (ferror nil (format:outfmt "Foo is " (format:onum foo)
                                   " which is too large"))
```

format:onum *number* &optional *radix minwidth* &key *padchar minpad tab-period signed commas*

Outputs *number* in base *radix*, padding to at least *minwidth* columns and obeying the other padding options specified as described above.

radix can be a number, or it can be :roman, :english, or :ordinal. The default *radix* is 10. (decimal).

If *signed* is non-nil, a + sign is printed if the number is positive. If *commas* is non-nil, a comma is printed every third digit in the customary way. These arguments are meaningful only with numeric radices.

format:ofloat *number* &optional *n-digits force-exponential-notation minwidth* &key *padchar minpad tab-period*

Outputs *number* as a floating point number using *n-digits* digits. If *force-exponential-notation* is non-nil, then an exponent is always used. *minwidth* and the padding options are interpreted as usual.

format:ostring *string* &optional *minwidth* &key *padchar minpad tab-period right-justify*

Outputs *string*, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the other padding options specified as described above.

Normally the data are left justified; any padding follows the data. If *right-justify* is non-nil, the padding comes before the data. The amount of padding is not affected.

The argument need not really be a string. Any Lisp object is allowed, and it is output with princ.

format:oprint *object* &optional *minwidth* &key *padchar minpad tab-period right-justify*

Prints *object*, any Lisp object, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the padding options specified as described above.

Normally the data are left justified; any padding follows the data. If *right-justify* is non-nil, the padding comes before the data. The amount of padding is not affected.

The printing of the object is done with prin1.

format:ochar *character* &optional *style top-explain minwidth* &key *padchar minpad tab-period*

Outputs *character* in one of three styles, selected by the *style* argument. *minwidth* and the padding options control padding as usual.

:read or nil	The character is printed using #\ or #/ so that it could be read back in.
:editor	Output is in the style of 'Meta-Rubout'. Non-printing characters, and the two printing characters Space and Altmode, are represented by their names. Other printing characters are printed directly.

:brief	Brief prefixes such as 'C-' and 'M-' are used, rather than 'Control-' or 'Meta-'. Also, character names are used only if there are meta bits present.
:lozenged	The output is the same as that of the :editor style, but If the character is not a graphic character or if it has meta bits, and the stream supports the :display-lozenged-string operation, that operation is used instead of :string-out to print the text. On windows this operation puts the character name inside a lozenge.
:sail	'α', '', etc. are used to represent Control and Meta, and shorter names for characters are also used when possible. See section 10.1.1, page 205.

top-explain is useful with the :editor, :brief and :sail styles. It says that any character that has to be typed using the Top or Greek keys should be followed by an explanation of how to type it. For example: '→ (Top-K)' or 'α (Greek-a)'.

format:tab *mincol* &key *padchar minpad tab-period terpri unit*

Outputs padding at least until column *mincol*. It is the only formatting function that bases its actions on the actual cursor position rather than the width of what is being output. The padding options *padchar*, *minpad*, and *tab-period* are obeyed. Thus, at least the *minpad* number of padding characters are output even if that goes past *mincol*, and once past *mincol*, padding can only stop at a multiple of *tab-period* characters past *mincol*.

In addition, if the *terpri* option is t, then if column *mincol* is passed, format:tab starts a new line and indents it to *mincol*.

The *unit* option specifies the units of horizontal position. The default is to count in units of characters. If *unit* is specified as :pixel, then the computation (and the argument *mincol* and the *minpad* and *tab-period* options) are in units of pixels.

format:pad (*minwidth* &key *padchar minpad tab-period...*) *body...* *Macro*

format:pad is used for printing several items in a fixed amount of horizontal space, padding between them to use up any excess space. Each of the *body* forms prints one item. The padding goes between items. The entire format:pad always uses at least *minwidth* columns; any columns that the items don't need are distributed as padding between the items. If that isn't enough space, then more space is allocated in units controlled by the *tab-period* option until there is enough space. If it's more than enough, the excess is used as padding.

If the *minpad* option is specified, then at least that many pad characters must go between each pair of items.

Padding goes only between items. If you want to treat several actual pieces of output as one item, put a progn around them. If you want padding before the first item or after the last, as well as between the items, include a dummy item nil at the beginning or the end.

If there is only one item, it is right justified. One item followed by **nil** is left-justified. One item preceded and followed by **nil** is centered. Therefore, **format:pad** can be used to provide the usual padding options for a function that does not provide them itself.

format:plural *number singular* &optional *plural*

Outputs either the singular or the plural form of a word depending on the value of *number*. The singular is used if and only if *number* is 1. *singular* specifies the singular form of the word. **string-pluralize** is used to compute the plural, unless *plural* is explicitly specified.

It is often useful for *number* to be a value returned by **format:onum**, which returns its argument. For example:

```
(format:plural (format:onum n-frobs) " frob")
```

prints "1 frob" or "2 frobs".

format:breakline *linel print-if-terpri print-always...* *Macro*

Goes to the next line if there is not enough room for something to be output on the current line. The *print-always* forms print the text which is supposed to fit on the line. *linel* is the column before which the text must end. If it doesn't end before that column, then **format:breakline** moves to the next line and executes the *print-if-terpri* form before doing the *print-always* forms.

Constant strings are allowed as well as forms for *print-if-terpri* and *print-always*. A constant string is just printed.

To go to a new line unconditionally, simply call **terpri**.

Here is an example that prints the elements of a list, separated by commas, breaking lines between elements when necessary.

```
(defun pcl (list linel)
  (do ((l list (cdr l))) ((null l))
    (format:breakline linel "  "
      (princ (car l))
      (and (cdr l) (princ ", ")))))
```

22.5 Rubout Handling

The rubout handler is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to allow the user to edit minor mistakes made during type-in. At the same time, it is not supposed to get in the way; input is to be seen by Lisp as soon as a syntactically complete form has been typed. The definition of 'syntactically complete form' depends on the function that is reading from the stream; for **read**, it is a Lisp expression.

Some interactive streams ('editing Lisp listeners') have a rubout handler that allows input to be edited with the full power of the ZWEI editor. (ZWEI is the general editor implementation on which Zmacs and ZMail are based.) Most windows have a rubout handler that apes ZWEI, implementing about twenty common ZWEI commands. The cold load stream has a simple rubout handler that allows just rubbing out of single characters, and a few simple commands like clearing the screen and erasing the entire input typed so far. All three kinds of rubout handler use the same protocol, which is described in this section. We also say a little about the most common of the three rubout handlers.
[Eventually some version of ZWEI will be used for all streams except the cold load stream]

The tricky thing about the rubout handler is the need for it to figure out when you are all done. The idea of a rubout handler is that you can type in characters, and they are saved up in a buffer so that if you change your mind, you can rub them out and type different characters. However, at some point, the rubout handler has to decide that the time has come to stop putting characters into the buffer and to let the function parsing the input, such as **read**, return. This is called *activation*. The right time to activate depends on the function calling the rubout handler, and may be very complicated (if the function is **read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently-defined reader macros do, and so on). Rubout handlers should not have to know how to parse the characters in the buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The rubout handler interface is organized so that the calling function can do all the parsing, while the rubout handler does all the handling of editing commands, and the two are kept completely separate.

The basic way that the rubout handler works is as follows. When an input function that reads characters from a stream, such as **read** or **readline** (but not **tyi**), is invoked with a stream which has **:rubout-handler** in its **:which-operations** list, that function "enters" the rubout handler. It then goes ahead **:tyi**'ing characters from the stream. Because control is inside the rubout handler, the stream echoes these characters so the user can see what he is typing. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the rubout handler it is also handing it the responsibility for echoing.) The rubout handler is also saving all these characters in a buffer, for reasons disclosed in the following paragraph. When the parsing function decides it has enough input, it returns and control "leaves" the rubout handler. This is the easy case.

If the user types a rubout, a **throw** is done out of all recursive levels of **read**, reader macros, and so forth, back to the point where the rubout handler was entered. Also the rubout is echoed by erasing from the screen the character which was rubbed out. Now the **read** is tried over again, re-reading all the characters that have not been rubbed out, not echoing them this time. When the saved characters have been exhausted, additional input is read from the user in the usual fashion.

The effect of this is a complete separation of the functions of rubout handling and parsing, while at the same time mingling the execution of these two functions in such a way that input is always activated at just the right time. It does mean that the parsing function (in the usual case, **read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have non-trivial side-effects, since it may be called multiple times.

If an error occurs while inside the rubout handler, the error message is printed and then additional characters are read. When the user types a rubout, it rubs out the error message as well as the character that caused the error. The user can then proceed to type the corrected expression; the input will be reparsed from the beginning in the usual fashion.

The rubout handler based on the ZWEI editor interprets control characters in the usual ZWEI way: as editing commands, allowing you to edit your buffered input.

The common rubout handler also recognizes a subset of the editor commands, including **Rubout**, **Control-F** and **Meta-F** and others. Typing **Help** while in the rubout handler displays a list of the commands. The kill and yank commands in the rubout handler use the same kill ring as the editor, so you can kill an expression in the editor and yank it back into a rubout handler with **Control-Y**, or kill an expression in the rubout handler with **Control-K** or **Clear-input** and yank it back in the editor. The rubout processor also keeps a ring buffer of most recent input strings (a separate ring for each stream), and the commands **Control-C** and **Meta-C** retrieve from this ring just as **Control-Y** and **Meta-Y** do from the kill ring.

When not inside the rubout handler, and when typing at a program that uses control characters for its own purposes, control characters are treated the same as ordinary characters.

Some programs such as the debugger allow the user to type either a control character or an expression. In such programs, you are really not inside the rubout handler unless you have typed the beginning of an expression. When the input buffer is empty, a control character is treated as a command for the program (such as, **Control-C** to continue in the debugger); when there is text in the rubout handler buffer, the same character is treated as a rubout handler command. Another consequence of this is that the message you get by typing **Help** varies, being either the rubout handler's documentation or the debugger's documentation.

To write a parsing function that reads with rubout handling, use **with-input-editing**.

with-input-editing (*stream options*) *body*... *Macro*

Invokes the rubout handler on *stream*, if *stream* supports it, and then executes *body*. *body* is executed in any case, within the rubout handler if possible. **rubout-handler** is non-nil while in *body* if rubout handling is in use.

options are used as the rubout handler options. If already within an invocation of the rubout handler, *options* are appended to the front of the options already in effect. This happens if a function which reads input using **with-input-editing**, such as **read** or **readline**, is called from the body of another **with-input-editing**. The **:norecursive** option can be used to cause the outer set of options to be completely ignored even when not overridden by new ones.

body's values are returned by **with-input-editing**. *body* ought to read input from *stream* and return a Lisp object that represents the input. It should have no nontrivial side effects aside from reading input from *stream* structure, as it may be aborted at any time it reads input and may be executed over and over.

If the **:full-rubout** option is specified, and the user types some input and rubs it all out, the **with-input-editing** form returns immediately. See **:full-rubout**, below.

If a preemptive command is input by the user, **with-input-editing** returns immediately with the values being as specified below under the **:command** and **:preemptable** options. *body* is aborted from its call to the **:tyi** operation, and the input read so far remains in the rubout handler editing buffer to be read later.

rubout-handler *Variable*

If control is inside the rubout handler in this process, the value is the stream on which rubout handling is being done. Otherwise, the value is **nil**.

:rubout-handler *options function* &rest *args* *Operation on streams*

Invokes the rubout handler on the stream, with *options* as the options, and parses by applying *function* to *args*. **with-input-editing** uses this operation.

:read-bp *Operation on streams*

This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It returns the index within the rubout handler buffer which parsing has reached.

:force-rescan *Operation on streams*

This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It causes parsing to start again immediately from the beginning of the buffer.

:rescanning-p *Operation on streams*

This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It returns **t** if parsing is now being done on input already in the buffer, **nil** if parsing has used up all the buffered input and the next character parsed will come from the keyboard.

Each option in the list of rubout handler options consists of a list whose first element is a keyword and whose remaining elements are the arguments of that keyword. Note that this is not the same format as the arguments to a typical function that takes keyword arguments; rather this is an alist of options. The standard options are:

(**:activation** *fn args...*)
: Activate if certain characters are typed in. When the user types an activation character, the rubout handler moves the editing pointer immediately to the end of the buffer and inserts the activation character. This immediately causes the parsing function to begin rescanning the input.

fn is used to test characters for being activators. It is called with an input character as the first arg (possibly a fixnum, possibly a character object) and *args* as additional args. If *fn* returns non-nil, the character is an activation. *fn* is not called for blips.

After the parsing function has read the entire contents of the buffer, it sees the activation character as a blip (:activation *char numeric-arg*) where *char* is the character that activated and *numeric-arg* is the numeric arg that was pending for the next rubout handler command. Normally the parsing function will return at this point. Then the activation character does not echo. But if the parsing function continues to read input, the activation character echoes and is inserted in the buffer.

(:do-not-echo *chars...*)
Poor man's activation characters. Like :activation except that the characters that should activate are listed explicitly, and the character itself is returned to the parsing function rather than a blip.

(:full-rubout *val*)
If the user rubs out all the characters he typed, then control is returned from the rubout handler immediately. Two values are returned; the first is nil and the second is *val*. (If the user doesn't rub out all the characters, then the rubout handler propagates multiple values back from the function that it calls, as usual.) In the absence of this option, the rubout handler would simply wait for more characters to be typed in and would ignore any additional rubouts.

This is how the debugger knows to remove Eval: from the screen if you type the beginning of a form and rub it all out.

(:pass-through *char1 char2...*)
The characters *char1*, *char2*, etc. are not to be treated as special by the rubout handler. They are read as input by the parsing function. If the parsing function does not return, they can be rubbed out. This works only for characters with no modifier bits.

(:preemptable *value*)
Makes all blips read as input by the rubout handler act as preemptive commands. If this option is specified, the rubout handler returns immediately when it reads a blip. It returns two values: the blip that was read, and *value*. The parsing function is not allowed to finish parsing up to a delimiter; instead, any buffered input remains in the buffer for the next time input is done. In the mean time, the preemptive command character can be processed by the command loop.

While this applies to all blips, the blips which it is probably intended for are mouse blips.

(:command *fn args...*)
Makes certain characters preemptive commands. A preemptive command returns instantly to the caller of the :rubout-handler operation, regardless

of the input in the buffer. It returns two values: a list (:command *char numeric-arg*) and the keyword :command. The parsing function is not allowed to finish parsing up to a delimiter; instead, any buffered input remains in the buffer for the next time input is done. In the mean time, the preemptive command character can be processed by the command loop.

The test for whether a character should be a preemptive command is done using *fn* and *args* just as in :activation.

(:editing-command (*char doc*)...)

Defines editing commands to be executed by the parsing function itself. This is how qsend implements the Control-Meta-Y command. Each *char* is such a command, and *doc* says what it does. (*doc* is printed out by the rubout handler's Help command.) If any of these characters is read by the rubout handler, it is returned immediately to the parsing function regardless of where the editing pointer is in the buffer. (Normal inserted text is not returned immediately when read unless the editing pointer is at the end of the buffer.)

The parsing function should not regard these characters as part of the input. There are two reasonable things that the parsing function can do when it receives one of the editing command characters: print some output, or force some input.

If it prints output, it should invoke the :refresh-rubout-handler operation afterward before the next :tyi. This causes the rubout handler to redisplay so that the input being edited appears after the output that was done.

If the parsing function forces input, the input is read by the rubout handler. This can be used to modify the buffered input. qsend's Control-Meta-Y command works by forcing the yanked text as input. There is no way to act directly on the buffered input because different implementations of the rubout handler store it in different ways.

(:prompt *function*)
(:reprompt *function*)

When it is time for the user to be prompted, *function* is called with two arguments. The first is a stream it may print on; the second is the character which caused the need for prompting, e.g. #\clear-input or #\clear-screen, or nil if the rubout handler was just entered.

The difference between :prompt and :reprompt is that the latter does not call the prompt function when the rubout handler is first entered, but only when the input is redisplayed (e.g. after a screen clear). If both options are specified then :reprompt overrides :prompt except when the rubout handler is first entered.

function may also be a string. Then it is simply printed.

If the rubout handler is exited with an empty buffer due to the **:full-rubout** option, whatever prompt was printed is erased.

(**:initial-input** *string*)
Pretends that the user typed *string*. When the rubout handler is entered, *string* is typed out. The user can input more characters or rub out characters from it.

(**:initial-input-index** *index*)
Positions the editing pointer initially *index* characters into the initial input string. Used only in company with with **:initial-input**.

(**:no-input-save t**)
Don't save this batch of input in the input history when it is done. For example, **yes-or-no-p** specifies this option.

(**:norecursive t**)
If this invocation of the rubout handler is within another one, the options specified in the previous call should be completely ignored during this one. Normally, individual options specified this time override the previous settings for the same options, but any of the previous options not individually overridden are still in effect.

Rubout handlers handle the condition **sys:parse-error** if it is signaled by the parsing function. The handling consists of printing the error message, waiting for the user to rub out, erasing the error message, and parsing the input again. All errors signaled by a parsing function that signify that the user's input was syntactically invalid should have this condition name. For example, the errors **read** signals have condition name **sys:parse-error** since it is is a consequence of **sys:read-error**.

sys:parse-error (error) *Condition*
The condition name for syntax errors in input being parsed.

The compiler handles **sys:parse-error** by proceeding with proceed-type **:no-action**. All signalers of **sys:parse-error** should offer this proceed type, and respond to its use by continuing to parse, ignoring the invalid input.

sys:parse-ferror *format-string* &rest *args*
Signals a **sys:parse-error** error, using *format-string* and *args* to print the error message. The proceed-type **:no-action** is provided, and if a handler uses it, this function returns **nil**.

23. Expression Input and Output

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*. This is what you have been seeing in the examples throughout this manual. 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. It and related functions are known as the *reader*. (Streams are explained in section 22.3, page 459.)

For the rest of the chapter, the phrase 'printed representation' is abbreviated as 'p.r.'

23.1 What the Printer Produces

The printed representation of an object depends on its type. In this section, we consider each type of object and explain how it is printed. There are several variables which you can set before calling the printer to control how certain kinds of objects print. They are mentioned where relevant in this section and summarized in the following section, but one of them is so important it must be described now. This is the *escaping* feature, controlled by the value of ***print-escape***.

Escaping means printing extra syntactical delimiters and *escape characters* when necessary to avoid ambiguity. Without escaping, a symbol is printed by printing the contents of its name; therefore, the symbol whose name consists of the three characters **1**, **.** and **5** prints just like the floating point number **1.5**. Escaping causes the symbol to print as **|1.5|** to differentiate the two. **|** is a kind of escape character; see page 516 for more information on escape characters and what they mean syntactically.

Escaping also involves printing package prefixes for symbols, printing double-quotes or suitable delimiters around the contents of strings, pathnames, host names, editor buffers, condition objects, and many other things. For example, without escaping, the pathname **SYS: SYS; QCP1 LISP** prints as exactly those characters. The string with those contents prints indistinguishably. With escaping, the pathname prints as

```
    #⊂FS:LOGICAL-PATHNAME "SYS: SYS; QCP1 LISP"⊃
```

and the string prints as **"SYS: SYS; QCP1 LISP"**.

The non-escaped version is nicer looking in general, but if you give it to **read** it won't do the right thing. The escaped version is carefully set up so that **read** will be able to read it in. Printing with escaping is useful in writing expressions into files. Printing without escaping is useful when constructing messages for the user. However, when the purpose of a message printed for the user is to *mention* an object, the object should be printed with escaping:

```
    Your output is in the file SYS: SYS; QCP1 QFASL.
vs
    Expected pathname properties missing from
     #⊂FS:LOGICAL-PATHNAME "SYS: SYS; QCP1 LISP"⊃.
```

The printed representation of an object also may depend on whether Common Lisp syntax is in use. Common Lisp syntax and traditional Zetalisp syntax are incompatible in some aspects of their specifications. In order to print objects so that they can be read back in, the printer needs to know which syntax rules the reader will use. This decision is based on the current readtable: the value of *readtable* at the time printing is done.

Now we describe how each type of object is standardly printed.

Integers:

For an integer (a fixnum or a bignum): the printed representation consists of

* a possible radix prefix
* a minus sign, if the number is negative
* the representation of the number's absolute value
* a possible radix suffix.

The radix used for printing the number's absolute value is found as the value of *print-base*. This should be either a positive fixnum or a symbol with an si:princ-function property. In the former case, the number is simply printed in that radix. In the latter case, the property is called as a function with two arguments, minus the absolute value of the number, and the stream to print on. The property is responsible for all printing. If the value of *print-base* is unsuitable, an error is signaled.

A radix prefix or suffix is used if either *nopoint is nil and the radix used is ten, or if *nopoint is non-nil and *print-radix* is non-nil. For radix ten, a period is used as the suffix. For any other radix, a prefix of the form #*radix*r is used. A radix prefix or suffix is useful to make sure that read parses the number using the same radix used to print it, or for reminding the user how to interpret the number.

Ratios:

The printed representation of a ratio consists of

* a possible radix prefix
* a minus sign, if the number is negative
* the numerator
* a ratio delimiter
* the denominator

If Common Lisp syntax is in use, the ratio delimiter is a slash (/). If traditional syntax is in use, backslash (\) is used. The numerator and denominator are printed according to *print-base*.

The condition for printing a radix prefix is the same as for integers, but a prefix #10r is used to indicate radix ten, rather than a period suffix.

Floating Point Numbers:

* a minus sign, if the number is negative
* one or more decimal digits
* a decimal point
* one or more decimal digits
* an exponent, if the number is small enough or large enough to require one. The exponent, if present, consists of
    * a delimiter, the letter e, s or f
    * a minus sign, if the exponent is negative

    one to three decimal digits

The number of digits printed is just enough to represent all the significant mantissa bits the number has. Feeding the p.r. of a float back to the reader is always supposed to produce an equal float. Floats are always printed in decimal; they are not affected by escaping or by *print-base*, and there are never any radix prefixes or suffixes.

The Lisp Machine supports two floating point number formats. At any time, one of them is the default; this is controlled by the value of *read-default-float-format*. When a floating point number whose format is *not* currently the default is printed, it must be printed with an exponent so that the exponent delimiter can specify the format. The exponent is introduced in this case by f or s to specify the format. To the reader, f specifies single-float format and s specifies short-float format.

A floating point number of the default format is printed with no exponent if this looks nice; namely, if this does not require too many extra zeros to be printed before or after the decimal point. Otherwise, an exponent is printed and is delimited with e. To the reader, e means 'use the default format'.

Normally the default float format is single-float. Therefore, the printer may print full size floats without exponents or with e exponents, but short floats are always printed with exponents introduced by s so as to tell the reader to make a short float.

Complex Numbers:

The traditional printed representation of a complex number consists of

* the real part
* a plus sign, if the imaginary part is positive
* the imaginary part
* the letter i, printed in lower case

If the imaginary part is negative, the + is omitted since the initial - of the imaginary part serves to separate it from the real part.

In Common Lisp syntax, a complex number is printed as #C(*realpart imagpart*); for example, #C(5 3). Common Lisp inexplicably does not allow the more natural 5+3i syntax.

The real and imaginary parts are printed individually according to the specifications above.

Symbols:

If escaping is off, the p.r. is simply the successive characters of the print-name of the symbol. If escaping is on, two changes must be made. First, the symbol might require a package prefix in order that **read** work correctly, assuming that the package into which **read** will read the symbol is the one in which it is being printed. See the chapter on packages (chapter 27, page 636) for an explanation of the package name prefix. If the symbol is one which would have another symbol substituted for it if printed normally and read back, such as the symbol **member** printed using Common Lisp syntax which would be replaced with **cli:member** if read in thus, it is printed with a package prefix (e.g., **global:member**) to make it read in properly. See page 519 for more information on this.

If the symbol is uninterned, #: is printed instead of a package prefix, provided ***print-gensym*** is non-nil.

Secondly, if the p.r. would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), then escape characters are added so as to suppress the other reading. Two kinds of escape characters may be used: single-character escapes and multiple escapes. A single-character escape can be used in front of a character to overrule its special syntactic meaning. Multiple escapes are used in pairs, and all the characters between the pair have their special syntactic meanings suppressed *except single-character escapes*. If the symbol name contains escape characters, they are escaped with single-character escapes. If the symbol name contains anything else problematical, a pair of multiple escape characters are printed around it.

The single-character and multiple escape characters are determined by the current readtable. Standardly the multiple escape character is vertical bar (|), in both traditional and Common Lisp syntax. The single-character escape character is slash (/) in traditional syntax and backslash (\) in Common Lisp syntax.

```
FOO             ;typical symbol, name composed of upper case letters
A/|B            ;symbol with a vertical bar in its name
|Symbol with lower case and spaces in its name|
|One containing slash (//) and vertical bar (/|) also|
```

Except when multiple escape characters are printed, any upper case letters in the symbol's name may be printed as lower case, according to the value of the variable ***print-case***. This is true whether escaping is enabled or not. See the next section for details.

Conses:

The p.r. for conses tends to favor *lists*. It starts with an open-parenthesis. Then the car of the cons is printed and the cdr of the cons is examined. If it is **nil**, a close-parenthesis is printed. If it is anything else but a cons, space dot space followed by that object is printed. If it is a cons, we print a space and start all over (from the point *after* we printed the open-parenthesis) using this new cons. Thus, a list is printed as an open-parenthesis, the p.r.'s of its elements separated by spaces, and a close-parenthesis. This is how the printer produces representations such as **(a b (foo bar) c)** in preference to synonymous forms such as **(a . (b . ((foo . (bar . nil)) . (c . nil))))**.

The following additional feature is provided for the p.r. of conses: as a list is printed, **print** maintains the length of the list so far and the depth of recursion of printing lists. If the length exceeds the value of the variable ***print-length***, **print** terminates the printed representation of the list with an ellipsis (three periods) and a close-parenthesis. If the depth of recursion exceeds the value of the variable ***print-level***, then the character **#** is printed instead of the list. These two features allow a kind of abbreviated printing that is more concise and suppresses detail. Of course, neither the ellipsis nor the **#** can be interpreted by **read**, since the relevant information is lost. In Common Lisp read syntax, either one causes **read** to signal an error.

If ***print-pretty*** is non-**nil**, conses are given to the grinder to print.

If ***print-circle*** is non-**nil**, a check is made for cars or cdrs that are circular or shared structure, and any object (except for an interned symbol) already mentioned is replaced by a ***#n#*** label reference. See page 524 for more information on them.

```
(let ((*print-circle* t))
  (prin1 (circular-list 3 4)))
```
prints
```
#1= (3 4 . #1#)
```

Character Objects:

When escaping is off, a character object is printed by printing the character itself, with no delimiters.

In Common Lisp syntax, a character object is printed with escaping as *#font\character-or-name*. *font* is the character's font number, in decimal, or is omitted if zero. *character-or-name* begins with prefixes for any modifier bits (control, meta, etc.) present in the character, each followed by a hyphen. Then comes a representation of the character sans font and modifier bits. If this reduced character is a graphic character, it represents itself. Otherwise, it certainly has a standard name; the name is used. If a graphic characters has special syntactic properties (such as whitespace, paretheses, and macro characters) and modifier bit prefixes have been printed then a single-character escape character is printed before it.

In traditional syntax, the p.r. is the similar except that the \ is replaced by `*/`.

Strings:

If escaping is off, the p.r. is simply the successive characters of the string. If escaping is on, double-quote characters ("") are printed surrounding the contents, and any single-character escape characters or double-quotes inside the contents are preceded by single-character escapes. If the string contains a Return character followed by an open parenthesis, a single-character escape is printed before the open parenthesis. Examples:

```
"Foo"
"/"Foo/", he said."
```

Named Structures:

If the named structure type symbol has a **named-structure-invoke** property, the property is called as a function with four arguments: the symbol **:print-self**, the named structure itself, the stream to print on, and the current *depth* of list structure (see below). It is this function's responsibility to output a suitable printed representation to the stream. This allows a user to define his own p.r. for his named structures; more information can be found in the named structure section (see page 390). Typically the printed representation used starts with either **#<** if it is not supposed to be readable or **#c** (see page 527) if it is supposed to be readable.

If the named structure symbol does not have a **named-structure-invoke** property, the printed-representation depends on whether escaping is in use. If it is, #s syntax is used:

```
#s ( named-structure-symbol
     component value
     component value
     ...)
```

Named structure component values are checked for circular or shared structure if ***print-circle*** is **non-nil**.

If escaping is off, the p.r. is like that used for miscellaneous data-types: **#<**, the named structure symbol, the numerical address of the structure, and **>**.

Other Arrays:

If ***print-array*** is non-**nil**, the array is printed in a way which shows the elements of the array. Bit vectors use `#*` syntax, other vectors use `#(...)` syntax, and arrays of rank other than one use `#na(...)` syntax. The printed representation does not indicate the array type (that is, what elements it is allowed to contain). If the printed representation is read in, a general array (array type **art-q**) is always created. See page 523 for more information on these syntaxes. Examples:

```
    (vector 1 2 5) => #(1 2 5)
    (make-array '(2 4) :initial-element t) => #2a((t t t t) (t t t t))
```

Vector and array groupings count like list groupings in maintaining the depth value that is compared with ***print-level*** for cutting off things that get too deep. More than ***print-length*** elements in a given vector or array grouping level are cut off with an ellipsis just like a list that is so long.

Array elements are checked for circular or shared structure if ***print-circle*** is non-**nil**.

If ***print-array*** is **nil**, the p.r. starts with **#<**. Then the **art-** symbol for the array type is printed. Next the dimensions of the array are printed, separated by hyphens. This is followed by a space, the machine address of the array, and a **>**, as in **#<ART-COMPLEX-FLOAT-3-6 34030451>**.

Instances and Entities:

If the object says it can handle the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure (see below), and whether escaping is enabled. The object should print a suitable p.r. on the stream. See chapter 21, page 401 for documentation on instances. Most such objects print like "any other data type" below, except with additional information such as a name. Some objects print only their name when escaping is not in effect (when **princ**'ed). Some objects, including pathnames, use a printed representation that begins with **#⊂**, ends with **⊃**, and contains sufficient information for the reader to reconstruct an equivalent object. See page 527. If the object cannot handle **:print-self**, it is printed like "any other data type".

Any Other Data Type:

The printed representation starts with **#<** and ends with **>**. This sort of printed representation cannot be read back in. The **#<** is followed by the **dtp-** symbol for this datatype, a space, and the octal machine address of the object. The object's name, if one can be determined, often appears before the address. If this style of printed representation is being used for a named structure or instance, other interesting information may appear as well. Finally a greater-than sign (**>**) is printed in octal. Examples:

```
    #'equal => #<DTP-U-ENTRY EQUAL 410>
    (value-cell-location nil) => #<DTP-LOCATIVE 1>
```

Including the machine address in the p.r. 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 will occasionally be moved, and therefore their octal machine addresses will be changed. It is best to shut off garbage collection temporarily when depending on these numbers.

Printed representations that start with '#<' can never be read back. This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. The following feature allows you to make sure that what you are printing may indeed be read with the reader.

si:print-readably *Variable*

When si:print-readably is bound to t, the printer signals an error if there is an attempt to print an object that cannot be interpreted by read. When the printer sends a :print-self or a :print message, it assumes that this error checking is done for it. Thus it is possible for these messages *not* to signal an error, if they see fit.

si:printing-random-object (*object stream . keywords*) &body *body* *Macro*

The vast majority of objects that define :print-self messages 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 si:print-readably. With no keywords, si:printing-random-object checks the value of si:print-readably and signals an error if it is not nil. It then prints a number sign and a less-than sign, evaluates the forms in *body*, then prints a space, the octal machine address of the object and a greater-than sign. A typical use of this macro might look like:

```
(si:printing-random-object (ship stream :typep)
  (tyo #\space stream)
  (prin1 (ship-name ship) stream))
```

This might print #<ship "ralph" 23655126>.

The following keywords may be used to modify the behaviour of si:printing-random-object:

:no-pointer	This suppresses printing of the octal address of the object.
:type	This prints the result of (type-of *object*) after the less-than sign. In the example above, this option could have been used instead of the first two forms in the body.

sys:print-not-readable (error) *Condition*

This condition is signaled by si:print-readably when the object cannot be printed readably.

The condition instance supports the operation :object, which returns the object that was being printed.

If you want to control the printed representation of some object, usually the right way to do it is to make the object an array that is a named structure (see page 390), or an instance of a flavor (see chapter 21, page 401). However, occasionally it is desirable to get control over all printing of objects, in order to change, in some way, how they are printed. If you need to do this, the best way to proceed is to customize the behavior of si:print-object (see page 543), which is the main internal function of the printer. All of 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 section 30.10, page 742).

23.2 Options that Control Printing

Several special variables are defined by the system for the user to set or bind before calling print or other printing functions. Their values, as set up by the user, control how various kinds of objects are printed.

print-escape *Variable*

Escaping is done if this variable is non-nil. See the previous section for a description of the many effects of escaping. Most of the output functions bind this variable to t or to nil, so you rarely use the variable itself.

print-base *Variable*
base *Variable*

The radix to use for printing integers and ratios. The value must be either an integer from 2 to 36 or a symbol with a valid si:princ-function property, such as :roman or :english.

The default value of *print-base* is ten. In input from files, the Base attribute (see section 25.5, page 594) controls the value of *print-base* (and of *read-base*).

The synonym base is from Maclisp.

print-radix *Variable*

If non-nil, integers and ratios are output with a prefix or suffix indicating the radix used to print them. For integers and radix ten, a period is printed as a suffix. Otherwise, a prefix such as #x or #3r is printed. The default value of *print-radix* is nil.

***nopoint** *Variable*

If the value of *nopoint is nil, a trailing decimal point is printed when a fixnum is printed out in base 10. This allows the numbers to be read back in correctly even if *read-base* is not 10 at the time of reading. The default value of *nopoint is t. *nopoint has no effect if *print-radix* is non-nil.

*nopoint exists for Maclisp compatibility. But to get truly compatible behavior, you must set *nopoint to nil (and, by default, base and ibase to eight).

print-circle *Variable*

If non-nil, the printer recognizes circular and shared structure and prints it using #*n*= labels so that it has a finite printed representation (which can be read back in). The default is nil, since t makes printing slower. See page 524 for information on the #*n*= construct.

print-pretty *Variable*

If non-nil, the printer actually calls grind-top-level so that it prints extra whitespace for the sake of formatting. The default is nil.

print-gensym *Variable*

If non-nil, uninterned symbols are printed with the prefix #: to mark them as such (but only when *print-escape* is non-nil). The prefix causes the reader to construct a similar uninterned symbol when the expression is read. If nil, no prefix is used for uninterned symbols. The default is t.

print-array *Variable*

If non-nil, non-string arrays are printed using the #(...), #* or #*n*a(...) syntax so that you can see their contents (and so that they can be read back in). If nil, such arrays are printed using #<...> syntax and do not show their contents. The default is nil. The printing of strings is not affected by this variable.

print-case *Variable*

Controls the case used for printing upper-case letters in the names of symbols. Its value should be :upcase, :downcase or :capitalize. These mean, respectively, to print those letters as upper case, to print them as lower case, or to capitalize each word (see string-capitalize, page 213). Any lower case letters in the symbol name are printed as lower case and escaped suitably; this flag does not affect them. Note that the case used for printing the upper case letters has no effect on reading the symbols back in, since they are case-converted by read. Any upper case letters that happen to be escaped are always printed in upper case.

```
(dolist (*print-case* '(:upcase :downcase :capitalize))
  (prin1-then-space 'foo)
  (prin1-then-space '|Foo|))
```

prints FOO |Foo| foo |Foo| Foo |Foo| .

print-level *Variable*
prinlevel *Variable*

print-level can be set to the maximum number of nested lists that can be printed before the printer gives up and just prints a # instead of a list element. If it is nil, which it is initially, any number of nested lists can be printed. Otherwise, the value of *print-level* must be a fixnum. Example:

```
(let ((*print-level* 2))
  (prin1 '(a (b (c (d e))))))
```

prints (a (b #)).

The synonym prinlevel is from Maclisp.

print-length *Variable*
prinlength *Variable*

print-length can be set to the maximum number of elements of a list that can be printed before the printer gives up and prints an ellipsis (three periods). If it is nil, which it is initially, any length list may be printed. Example:

```
(let ((*print-length* 3))
  (prin1 '((a b c d) #(e f g h) (i j k l) (m n o p))))
```

prints ((a b c ...) #(e f g ...) (i j k ...) ...).

The synonym **prinlength** is from Maclisp.

23.3 What The Reader Accepts

The purpose of the reader is to accept characters, interpret them as the p.r. of a Lisp object, and create and return such an object. The reader cannot accept everything that the printer produces; for example, the p.r.'s of compiled code objects, closures, stack groups, etc., 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.

This section shows what kind of p.r.'s the reader understands, and explains the readtable, reader macros, and various features provided by **read**.

The syntax specified for Common Lisp is incompatible with the traditional Zetalisp syntax. Therefore, the Lisp Machine supports both traditional and Common Lisp syntax, but **read** must be told in advance which one to use. This is controlled by the choice of readtable (see section 23.6, page 535). When reading input from a file, the Lisp system chooses the syntax according to the file's attribute list: Common Lisp syntax is used if the **Common Lisp** attribute is present (see section 25.5, page 594).

The main difference between traditional and Common Lisp syntax is that traditionally the single-character escape is slash (**/**), whereas in Common Lisp syntax it is backslash (\). Thus, the division function which in traditional syntax is written **//** is written just **/** in Common Lisp syntax. The other differences are obscure and are mentioned below where they occur.

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 p.r. of an atomic object such as a symbol or number, or a special character such as a parenthesis. The reader reads one or more tokens until the complete p.r. of an object has been seen, then constructs and returns that object.

Escape characters can be used to suppress the special syntactic significance of any character, including **:**, **Space**, **(** or **"**. There are two kinds of escape character: the *single-character escape* (**/** in traditional syntax, \ in Common Lisp syntax) suppresses the significance of the immediately following character; *multiple escapes* (vertical bar, **|**) are used in pairs, and suppress the special significance of all the characters except escapes between the pair. Escaping a character causes it to be treated as a token constituent and causes the token containing it to be read as a symbol. For example, **(12 5 x)** represents a list of three elements, two of which are integers, but **(/12 5/ x)** or **(|15| |5 X|)** represents a list of two elements, both symbols. Escaping also prevents conversion of letters to upper case, so that **|x|** is the symbol whose print name contains a lower-case **x**.

The circle-cross (**⊗**) character an *octal escape character* which may be useful for including weird characters in the input. The next three characters are read and interpreted as an octal number, and the character whose code is that number replaces the circle-cross and the digits in the input stream. This character is always treated as a token constituent and forces the token to be read as a symbol. **⊗** is allowed in both traditional and Common Lisp syntax, but it is not valid Common Lisp.

Integers:

The reader understands the p.r.'s of integers in a way more general than is employed by the printer. Here is a complete description of the format for integers.

Let a *simple integer* be a string of digits, optionally preceded by a plus sign or a minus sign, and optionally followed by a trailing decimal point. A simple integer is interpreted by **read** as an integer. If the trailing decimal point is present, the digits are interpreted in decimal radix; otherwise, they are considered as a number whose radix is the value of the variable ***read-base***.

read-base *Variable*
ibase *Variable*

The value of **ibase** or ***read-base*** is an integer between 2 and 36 that is the radix in which integers and ratios are read. The initial value of is ten. For input from files or editor buffers, the **Base** attribute specifies the value to be used (see section 25.5, page 594); if it is not given, the ambient value is used.

The synonym **ibase** is from Maclisp.

If the input radix is greater than ten, letters starting with **a** are used as additional "digits" with values ten and above. For example, in radix 16, the letters a through f are digits with values ten through 15. Alphabetic case is not significant. These additional digits can be used wherever a simple integer is expected and are parsed using the current input radix. For example, if ***read-base*** is 16 then `ff` is recognized as an integer (255 decimal). So is **10e5**, which is a float when ***read-base*** is ten.

Traditional syntax also permits a simple integer, followed by an underscore (_) or a circumflex (^), followed by another simple integer. The two simple integers are interpreted in the usual way; the character in between indicates an operation that is then performed on the two integers. The underscore indicates a binary "left shift"; that is, the integer to its left is doubled the number of times indicated by the integer to its right. The circumflex multiplies the integer to its left by ***read-base*** the number of times indicated by the integer to its right. (The second simple integer is not allowed to have a leading minus sign.) Examples: **3_2** means **12** and **645^3** means **645000**.

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

```
4
23456.
-546
+45^+6      ;means 45000000
2_11        ;4096
72361356126536125376512375126535123712635
-123456789.
105_1000    ;(ash 105 1000) has this value.
105_1000.
```

Floating Point Numbers:

Floats can be written with or without exponent. The syntax for a float without exponent is an optional plus or minus sign, optionally some digits, a decimal point, and one or more digits. A float with exponent consists of a simple integer or a float without exponent, followed by an exponent delimiter (a letter) and a simple integer (the exponent itself) which is the power of ten by which the number is to be scaled. The exponent may not have a trailing decimal point. Both the mantissa and the exponent are always interpreted in base ten, regardless of the value of ***read-base***.

Only certain letters are allowed for delimiting the exponent: e, s, f, d and l. The case of the letter is not significant. s specifies that the number should be a short float; f, that it should be a full-size float. d or l are equivalent to f; Common Lisp defines them to mean 'double float' or 'long float', but the Lisp Machine does not support anything longer than a full-size float, so it regards d and l as synonymous with f. e tells the reader to use the current default format, whatever it may be, as specified by the value of ***read-default-float-format***.

read-default-float-format *Variable*

The value is the type for read to produce by default for floats whose precise type is not specified by the syntax. The value should be either **global:small-float** or **global:single-float**, these being the only distinct floating formats that the Lisp Machine has. The default is **single-float**, to make full-size floats.

Here are some examples of printed-representations that always read as full-size floats:

```
6.03f23  1F-9     1.f3     3d6
```

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

```
0s0      1.5s9    -42S3    1.s5
```

These read as floats or as a short floats according to ***read-default-float-format***:

```
0.0      1.5      14.0     0.01
.707     -.3      +3.14159 6.03e23
1E-9     1.e3
```

Rationals:

The syntax for a rational is an integer, a ratio delimiter, and another integer. The integers may not include the ^ and _ scaling characters or decimal points, and only the first one may have a sign. The ratio delimiter is backslash (\) in traditional syntax, slash (/) in Common Lisp syntax. Here are examples:

1\2	-100000000000000\3	80\10	traditional
1/2	-100000000000000/3	80/10	Common Lisp

Recall that rationals include the integers; 80\10 as input to the reader is equivalent to 8.

Complex Numbers:

The traditional syntax for a complex number is a number (for the real part), a sign (+ or -), an unsigned number (for the imaginary part), and the letter i. The real and imaginary parts can be any type of number, but they are converted to be of the same type (both floating of the same format, or both rational). For example:

```
1-3\4i
1.2s0+3.45s8i
```

The Common Lisp syntax for a complex number is `#c(`*real imag*`)`, where *real* is the real part and *imag* is the imaginary part. This construction is allowed in traditional syntax too.

```
#c(1 -3/4)
#c(1.2s0 3.45s8)
```

Symbols:

A string of letters, numbers, and characters without special syntactic meaning is recognized by the reader as a symbol, provided it cannot be interpreted as a number. Alphabetic case is ignored in symbols; lower-case letters are translated to upper-case unless escaped. When the reader sees the p.r. of a symbol, it *interns* it on a *package* (see chapter 27, page 636, for an explanation of interning and the package system). Symbols may start with digits; you could even have one named **-345t**; **read** accepts this as a symbol without complaint. If you want to put strange characters (such as lower-case letters, parentheses, or reader macro characters) inside the name of a symbol, they must be escaped. If the symbol's name would look like a number, at least one character in the name must be escaped, but it matters not which one.
Examples of symbols:

```
foo
bar/(baz/)      ; traditional
bar\(baz\)      ; Common Lisp
34w23
|Frob Sale| and F|rob |S|ale| are equivalent

|a/|b|     ; traditional
|a\|b|     ; Common Lisp
```

In Common Lisp syntax, a symbol composed only of two or more periods is not allowed unless escaping is used.

The reader can be directed to perform substitutions on the symbols it reads. Symbol substitutions are used to implement the incompatible Common Lisp definitions of various system functions. Reading of Common Lisp code is done with substitutions that replace **subst** with **cli:subst**, **member** with **cli:member**, and so on. This is why, when a Common Lisp program uses the function **member**, it gets the standard Common Lisp **member** function rather than the traditional one. This is why we say that **cli:member** is "the Common Lisp version of **member**". While **cli:member** can be referred to from any program in just that way, it exists primarily to be referred to from a Common Lisp program which says simply **member**.

Symbol substitutions do not apply to symbols written with package prefixes, so one can use a package prefix to force a reference to a symbol that is normally substituted for, such as using **global:member** in a Common Lisp program.

Strings:

Strings are written with double-quote characters (") before and after the string contents. To include a double-quote character or single-character escape character in the contents, write an extra single-character escape character in front of it.
Examples of strings:

```
"This is a typical string."
"That is a /"cons cell/"."     ;; traditional
"That is a \"cons cell\"."     ;; Common Lisp
"Strings are often used for I//O."     ;; traditional
"Strings are often used for I/O."      ;; Common Lisp
"Here comes one backslash: \\"         ;; Common Lisp
```

Conses:

When **read** sees an open-parenthesis, it knows that the p.r. of a cons is coming, and calls itself recursively to get the elements of the cons or the list that follows. The following are valid p.r.'s of conses:

```
(foo . bar)
(foo "bar" 33)
(foo . ("bar" . (33 . nil)))
(foo bar . quux)
```

The first is a cons, whose car and cdr are both symbols. The second is a list, and the third is equivalent to the second (although **print** would never produce it). The fourth is a dotted list; the cdr of the last cons cell (the second one) is not **nil**, but **quux**.

The reader always allocates new cons cells to represent parentheses. They are never shared with other structure, not even part of the same read. For example,

```
(let ((x (read)))
  (eq (car x) (cdr x)))
((a b) . (a b))          ;; data for read
  => nil
```

because each time **(a b)** is read, a new list is constructed. This contrasts with the case for symbols, as very often **read** returns symbols that it found interned in the package rather than creating new symbols itself. Symbols are the only thing that work this way.

The dot that separates the two elements of a dotted-pair p.r. for a cons is only recognized if it is surrounded by delimiters (typically spaces). Thus dot may be freely used within print-names of symbols and within numbers. This is not compatible with Maclisp; in Maclisp **(a.b)** reads as a cons of symbols **a** and **b**, whereas in Zetalisp it reads as a list of a symbol **a.b**.

Comments:

A comment begins with a semicolon (;) and continues to the end of the line. Comments are ignored completely by the reader. If the semicolon is escaped or inside a string, it is not recognized as starting a comment; it is part of a symbol or part of the string.

```
;; This is a comment.
"This is a string; but no comment."
```

Another way to write a comment is to start it with `#|` and end it with `|#`. This is useful for commenting out multiple-line segments of code. The two delimiters nest, so that `#| #| |# |#` is a single comment. This prevents surprising results if you use this construct to comment out code which already contains such a comment.

```
(cond ((atom x) y)
 #|
      ((foo x)
       (do-it y))
    |#
      (t (hack y)))
```

Abbreviations:

The single-quote character (') is an abbreviation for a list starting with the symbol **quote**. The following pairs of p.r.'s produce equal lists:

```
'a     and   (quote a)
'(x (y))   and   (quote (x (y)))
```

The backquote character (`) and comma are used in a syntax that abbreviates calls to the list and vector construction functions. For example,

```
`(a ,b c)
```

reads as a list whose meaning as a Lisp form is equivalent to

```
(list 'a b 'c)
```

See section 18.2.2, page 325 for full details about backquote.

23.3.1 Sharp-sign Constructs

Sharp-sign (`#`) is used to introduce syntax extensions. It is the beginning of a two-character sequence whose meaning depends on the second character. Sharp-sign is only recognized with a special meaning if it occurs at the beginning of a token. If encountered while a token is in progress, it is a symbol constituent. For example, `#xff` is a sharp-sign construct that interprets `ff` as a hexidecimal number, but `1#xff` is just a symbol.

If the sharp-sign is followed by decimal digits, the digits form a parameter. The first non-digit determines which sharp-sign construct is actually in use, and the decimal integer parsed from the digits is passed to it. For example, `#r` means "read in specified radix"; it must actually be used with a radix notated in decimal between the `#` and the `r`, as in `#8r`.

It is possible for a sharp-sign construct to have different meanings in Common Lisp and traditional syntax. The only constructs which differ are **#** and **#/**.

The function **set-dispatch-macro-character** (see page 541) can be used to define additional sharp sign abbreviations.

Here are the currently-defined sharp sign constructs:

#/ #/ is used in traditional syntax only to represent the number that is the character code for a character. You can follow the **#/** with the character itself, or with the character's name. The name is preferable for nonprinting characters, and it is the only way to represent characters which have control bits since they cannot go in files. Here are examples of **#/**:

`#/a`	`#o141`
`#/A`	`#o101`
`#/(`	`#o50`
`#/c-a`	the character code for **Control-A**
`#/c-/a`	the character code for **Control-a**
`#/c-sh-a`	the character code for **Control-a**
`#/c-/A`	the character code for **Control-A**
`#/c-/(`	the character code for **Control-(**
`#/return`	the character code for **Return**
`#/h-m-system`	the character code for **Hyper-Meta-System**

To represent a printing character, write **#/**_x_ where _x_ is the character. For example, **#/a** is equivalent to **#o141** but clearer in its intent. To avoid ambiguity, the character following _x_ should not be a letter; good style would require this anyway.

As in strings, upper and lower-case letters are distinguished after **#/**. Any character works after **#/**, even those that are normally special to **read**, such as parentheses. Thus, **#/A** is equivalent to **#o101**, and **#/(** is equivalent to **#o50**. Note that the slash causes this construct to be parsed correctly by the editors Emacs and Zmacs. Even non-printing characters may be used, but for them it is preferable to use the character's name.

To refer to a character by name, write **#/** followed by the name. For example, **#/return** reads as the numeric code for the character **Return**. The defined character names are documented below (see section 10.1.6, page 211). In general, the names that are written on the keyboard keys are accepted. In addition, all the nonalphanumeric characters have names. The abbreviations **cr** for **return** and **sp** for **space** are accepted, 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 letters are converted to upper case unless escaped, 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 that **#/** uses; therefore, any character name typed out is acceptable as input.

#/ can also be used to read in the names of characters that have modifier bits (Control, Meta, Super and Hyper). 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 control, meta, hyper, and super. They may be in any order, and case is not significant. Prefix bit names can be abbreviated as the single letters c, m, h and s, and control may be spelled ctrl as it is on the keyboard. The last hyphen may be followed by a single character or by any of the special character names normally recognized by #/. A single character is treated the same way the reader normally treats characters in symbols; if you want to use a lower-case character or a special character such as a parenthesis, you must precede it by a slash character. Examples: #/Hyper-Super-A, #/meta-hyper-roman-i, #/CTRL-META-/(.

An obsolete method of specifying control bits in a character is to insert the characters α, β, ε, π and λ between the # and the /. Those stand for control, meta, control-meta, super and hyper, respectively. This syntax should be converted to the new #\control-meta-x syntax described below.

greek (or front), top, and shift (or sh) are also allowed as prefixes of names. Thus, #/top-g is equivalent to #/↑ or #/uparrow. #/top-g should be used if you are specifying the keyboard commands of a program and the mnemonic significance belongs to the 'G' rather than to the actual character code.

#\ In traditional syntax, #\ is a synonym for #/. In the past, #/ had to be used before a single character and #\ had to be used in all other cases. Now either one is allowed in either case.

In Common Lisp syntax, #\ produces a character object rather than a fixnum representing a character.

#≠/ #≠/*x* is the traditional syntax way to produce a character object. It is used just like #/. Thus, Common Lisp #\ is equivalent to traditional syntax #≠/.

#^ #^*x* is exactly like #/control-*x* if the input is being read by Zetalisp; it generates Control-*x*. In Maclisp *x* is converted to upper case 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*. (In Maclisp #/control-*x* sets the bit set by the Control key when the TTY is open in fixnum mode.)

#' #'*foo* is an abbreviation for (function *foo*). *foo* is the p.r. of any object. This abbreviation can be remembered by analogy with the ' macro-character, since the function and quote special forms are somewhat analogous.

#(#(*elements*...) constructs a vector (rank-one array) of type art-q with elements *elements*. The length of the vector is the number of elements written. Thus, #(a 5 "Foo") reads as a vector containing a symbol, an integer and a string. If a decimal integer appears after the #, it specifies the length of the vector. The last element written is replicated to fill the remaining elements.

#a #*n*a *contents* signifies an array of rank *n*, containing *contents*. *contents* is passed to make-array as the *initial-contents* argument. It is a list of lists of lists... or vector of vectors... as deep as *n*. The dimensions of the array are specified by the lengths of the lists or vectors. The rank is specified explicitly so that the reader can distinguish whether

a list or vector in the contents is a sequence of array elements or a single array element. The array type is always **art-q**.
Examples:

```
        #2a ((x y) (a b) ((uu 3) "VV"))
```

produces a 3 by 2 array. **(uu 3)** is one of the elements.

```
        #2a ("foo" "bar")
```

produces a 2 by 3 array whose elements are character objects. Recall that a string is a kind of vector.

```
        #0a 5
```

produces a rank-0 array whose sole element is 5.

#* #**bbb*...* signifies a bit vector; *bbb...* are the bits (characters **1** or **0**). A vector of type **art-1b** is created and filled with the specified bits, the first bit specified going in array element 0. The length is however many bits you specify. Alternatively, specify the length with a decimal number between the # and the *. The last **1** or **0** specified is duplicated to fill the additional bits. Thus, **#8*0101** is the same as **#*01011111**.

#s #s(*type slot value slot value slot value* ...) constructs a structure of type *type*. Any structure type defined with **defstruct** can be used as *type* provided it has a standard constructor taking slot values as keyword arguments. (Standard constructors can be functions or macros; either kind works for **#s**.) The slot names and values appearing in the read syntax are passed to the constructor so that they initialize the structure. Example:

```
        (defstruct (foo :named)
          bar
          lose)
        #s (foo :bar 5 :lose haha)
```

produces a **foo** whose **bar** component is 5 and whose **lose** component is **haha**.

#=
Are used to represent circular structure or shared structure. #*n*= preceding an object "labels" that object with the label *n*, a decimal integer. This has no effect on the way the object labeled is read, but it makes the label available for use in a #*n*# construct within that object (to create circular structure) or later on (to create shared structure). #*n*# counts as an object in itself, and reads as the object labeled by *n*.

For example, **#1=(a . #1#)** is a way of notating a circular list such as would be produced by **(circular-list 'a)**. The list is labeled with label 1, and then its cdr is given as a reference to label 1. **(#1=#:foo #1#)** is an example of shared structure. An uninterned symbol named **foo** is used as the first element of the list, and labeled. The second element of the list is the very same uninterned symbol, by virtue of a reference to the label.

Printing outputs #*n*= and #*n*# to represent circular or shared structure when ***print-circle** is non-**nil**.

#, Evaluate a form at load time. #, *foo* evaluates *foo* (the p.r. of a Lisp form) at read time, except that during file-to-file compilation it is arranged that *foo* will be evaluated when the QFASL 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 if you want it, 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 p.r. of a lisp form) at read time, regardless of who is doing the reading.

#' #' is a construct for repeating an expression with some subexpressions varying. It is an abbreviation for writing several similar expressions or for the use of **mapc**. Each subexpression that is to be varied is written as a comma followed by a list of the things to substitute. The expression is expanded at read time into a **progn** containing the individual versions.

```
        #'(send stream ',(:clear-input :clear-output))
```
expands into
```
        (progn (send stream :clear-input)
               (send stream :clear-output))
```

Multiple repetitions can be done in parallel by using commas in several subexpressions:

```
        #'(rename-file ,("foo" "bar") ,("ofoo" "obar"))
```
expands into
```
        (progn (rename-file "foo" "ofoo")
               (rename-file "bar" "obar"))
```

If you want to do multiple independent repetitions, you must use nested #' constructs. Individual commas inside the inner #' apply to that #'; they vary at maximum speed. To specify a subexpression that varies in the outer #', use two commas.

```
        #'#'(print (* ,(5 7) ,,(11. 13.)))
```
expands into
```
        (progn (progn (print (* 5 11.)) (print (* 7 11.)))
               (progn (print (* 5 13.)) (print (* 7 13.))))
```

#o #o *number* reads *number* in octal regardless of the setting of ***read-base***. Actually, any expression can be prefixed by #o; it is read with ***read-base*** bound to 8.

#b Like #o but reads in binary.

#x Like #x but reads in radix 16 (hexadecimal). The letters a through f are used as the digits beyond 9.

#r *#radix*r *number* reads *number* in radix *radix* regardless of the setting of ***read-base***. As with #o, any expression can be prefixed by *#radix*r; it is read with ***read-base*** bound to *radix*. *radix* must be a valid decimal integer between 2 and 36.

For example, #3r102 is another way of writing 11. and #11r32 is another way of writing 35. Bases larger than ten use the letters starting with a as the additional digits.

#c #c(*real imag*) constructs a complex number with real part *real* and imaginary *part*. It is equivalent to *real*+*imag*i, except that #c is allowed in Common Lisp syntax and the other is not.

#+ This abbreviation provides a read-time conditionalization facility. It is used as *#+feature form*. If *feature* is a symbol, then this is read as *form* if *feature* is present in the list ***features*** (see page 803). Otherwise, the construct is regarded as whitespace.

Alternately, *feature* may be a boolean expression composed of and, or, and not operators and symbols representing items that may appear on *features*. Thus, #+(or lispm amber) causes the following object to be seen if either of the features lispm or amber is present.

For example, #+lispm *form* makes *form* count if being read by Zetalisp, and is thus equivalent to #q *form*. Similarly, #+maclisp *form* is equivalent to #m *form*. #+(or lispm nil) *form* makes *form* count on either Zetalisp or in NIL.

Here is a list of features with standard meanings:

lispm	This feature is present on any Lisp machine (no matter what version of hardware or software).
maclisp	This feature is present in Maclisp.
nil	This feature is present in NIL (New Implementation of Lisp).
mit	This feature is present in the MIT Lisp machine system, which is what this manual is about.
symbolics	This feature is present in the Symbolics version of the Lisp machine system. May you be spared the dishonor of using it.

#+, and the other read-time conditionalization constructs that follow, discard the following expression by reading it with *read-suppress* bound to t if the specified condition is false.

#- #-*feature form* is equivalent to #+(not *feature*) *form*.

#q #q *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace). This is considered obsolete; use #+lispm instead.

#m #m *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace). This is considered obsolete; use #+maclisp instead.

#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 (white space). This is considered obsolete; use #+nil instead.

#◊ #◊ introduces an expression in infix notation. ◊ should be used to terminate it. The text in between describes a Lisp object such as a symbol, number or list but using a nonstandard, infix-oriented syntax. For example,

```
        #◊x:y+car(a1[i,j])◊
```

is equivalent to

```
        (setq x (+ y (car (aref a1 i j))))
```

It is not strictly true that the Lisp object produced in this way has to be an expression. Since the conversion is done at read time, you can use a list expressed this way for any purpose. But the infix syntax is designed to be used for expressions.

For full details, refer to the file SYS: IO1; INFIX LISP.

#< This is not legal reader syntax. It is used in the p.r. of objects that cannot be read back in. Attempting to read a #< signals an error.

#⊂
This is used in the p.r. of miscellaneous objects (usually named structures or instances) that can be read back in. #⊂ should be followed by a typename and any other data needed to construct an object, terminated with a ⊃. For example, a pathname might print as

```
        #⊂FS:ITS-PATHNAME "AI: RMS; TEST 5"⊃
```

The typename is a keyword that **read** uses to figure out how to read in the rest of the printed representation and construct the object. It is read in in package **user** (but it can contain a package prefix). The resulting symbol should either have a **si:read-instance** property or be the name of a flavor that handles the **:read-instance** operation.

In the first case, the property is applied as a function to the typename symbol itself and the input stream. In the second, the handler for that operation is applied to the operation name (as always), the typename symbol, and the input stream (three arguments, but the first is implicit and not mentioned in the **defmethod**). **self** will be **nil** and instance variables should not be referred to. **si:print-readably-mixin** is a useful implementation the **:read-instance** operation for general purposes; see page 446.

In either case, the handler function should read the remaining data from the stream, and construct and return the datum it describes. It should return with the ⊃ character waiting to be read from the input stream (**:untyi** it if necessary). **read** signals an error after it is returned to if a ⊃ character is not next.

The typename can be any symbol with an appropriate property or flavor, not necessarily related to the type of object that is created; but for clarity, it is good if it is the same as the **type-of** of the object printed. Since the type symbol is passed to the handler, one flavor's handler can be inherited by many other flavors and can examine the type symbol read in to decide what flavor to construct.

#| #| is used to comment out entire pieces of code. Such a comment begins with #| and ends with |#. The text in between should be one or more properly balanced p.r.'s of Lisp objects, possibly including nested #|...|# comments. This text is skipped over by the reader, and does not contribute to the value returned by **read**.

23.4 Expression Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of ***standard-output***. If *stream* is **nil**, the value of ***standard-output*** (i.e. the default) is used. If it is **t**, the value of ***terminal-io*** is used (i.e. the interactive terminal). This is all more-or-less compatible with Maclisp, except that instead of the variable ***standard-output*** Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

prin1 *object* &optional *stream*
Outputs the printed representation of *object* to *stream*, with escaping (see page 506). *object* is returned.

prin1-then-space *object* &optional *stream*
Like **prin1** except that output is followed by a space.

print *object* &optional *stream*
Like **prin1** except that output is preceded by a carriage return and followed by a space. *object* is returned.

princ *object* &optional *stream*
Like **prin1** except that the output is not escaped. *object* is returned.

write *object* &key *stream escape radix base circle pretty level length case gensym array*
Prints *object* on *stream*, having bound all the printing flags according to the keyword arguments if specified. For example, the keyword argument *array* specifies how to bind ***print-array***; if *array* is omitted, the ambient value of ***print-array*** is used. This function is sometimes cleaner than binding a printing control variable explicitly. The value is *object*.

23.4.1 Pretty-Printing Output Functions

pprint *object* &optional *stream*
pprint is like **prin1** except that ***print-pretty*** is bound to **t** so that the grinder is used. **pprint** returns zero values, just as the form **(values)** does.

grindef *function-spec...* *Macro*
Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed: The **quote** special form is represented with the ' character. Displacing macros are printed as the original code rather than the result of macro expansion. The code resulting from the backquote (**`**) reader macro is represented in terms of **`**.

The subforms to **grindef** are the function specs whose definitions are to be printed; the usual way **grindef** is used is with a form like **(grindef foo)** to print the definition of **foo**. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as **defun** special forms, and values are printed as **setq** special forms.

If a function is compiled, **grindef** says so and tries to find its previous interpreted definition by looking on an associated property list (see **uncompile** (page 301). This works only if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a QFASL file, **grindef** cannot not find the interpreted definition.

With no subforms, **grindef** assumes the same arguments as when it was last called.

grind-top-level *obj* &optional *width* (*stream* *standard-output*) (*untyo-p* nil) (*displaced* 'si:displaced) (*terpri-p* t) *notify-fun* *loc*

Pretty-prints *obj* on *stream*, putting up to *width* characters per line. This is the primitive interface to the pretty-printer. Note that it does not support variable-width fonts. If the *width* argument is supplied, it is how many characters wide the output is to be. If *width* is unsupplied or nil, grind-top-level tries to figure out the natural width of the stream, by sending a :size-in-characters message to the stream and using the first returned value. If the stream doesn't handle that message, a width of 95. characters is used instead.

The remaining optional arguments activate various strange features and usually should not be supplied. These options are for internal use by the system and are documented here for only completeness. If *untyo-p* is t, the :untyo and :untyo-mark operations are be used on *stream*, speeding up the algorithm somewhat. *displaced* controls the checking for displacing macros; it is the symbol which flags a place that has been displaced, or nil to disable the feature. If *terpri-p* is nil, grind-top-level does not advance to a fresh line before printing.

If *notify-fun* is non-nil, it should be a function that to be called with three arguments for each "token" in the pretty-printed output. Tokens are atoms, open and close parentheses, and reader macro characters such as '. The arguments given to *notify-fun* are the token, its "location" (see next paragraph), and t if it is an atom or nil if it is a character.

loc is the "location" (typically a cons) whose car is *obj*. As the grinder recursively descends through the structure being printed, it keeps track of the location where each thing came from, for the benefit of the *notify-fun*. This makes it possible for a program to correlate the printed output with the list structure. The "location" of a close parenthesis is t, because close parentheses have no associated location.

23.4.2 Non-Stream Printing Functions

write-to-string *object* &key *escape radix base circle pretty level length case gensym array*
prin1-to-string *object*
princ-to-string *object*

Like write, prin1 and princ, respectively, but put the output in a string and return the string (see page 528).

See also the with-output-to-string special form (page 474).

The following obsolete functions are for Maclisp compatibility only. The examples use traditional syntax.

exploden *object*

Returns a list of characters (represented as fixnums) that are the characters that would be typed out by (princ *object*) (i.e. the unescaped printed representation of *object*).

Example:

```
        (exploden '(+ /12 3)) => #o(50 53 40 61 62 40 63 51)
```

explodec *object*

Returns a list of characters represented by symbols, interned in the current package, whose names are the characters that would be typed out by (princ *object*) (i.e. the unescaped printed representation of *object*).

Example:

```
        (explodec '(+ /12 3)) => (|(| + | | |1| |2| | | |3| |)|)
```

(Note that there are escaped spaces in the above list.)

explode *object*

Like explodec but uses the escaped printed representation.

Example:

```
        (explode '(+ /12 3)) => (|(| + | | // |1| |2| | | |3| |)|)
```

(Note that there are escaped spaces in the above list.)

flatsize *object*

Returns the number of characters in the escaped printed representation of *object*.

flatc *object*

Returns the number of characters in the unescaped printed representation of *object*.

23.5 Expression Input Functions

Most expression input functions read characters from an input stream. This argument is called *stream*. If unsupplied it defaults to the value of *standard-input*.

All of these functions echo their input and permit editing if used on an interactive stream (one which supports the :rubout-handler operation; see below.)

The functions accept an argument *eof-option* or two arguments *eof-error* and *eof-value* to tell them what to do if end of file is encountered instead of an object's p.r. The functions that take two *eof-* arguments are the Common Lisp ones.

In functions that accept the *eof-option* argument, if no argument is supplied, an error is signaled at eof. If the argument is supplied, end of file causes the function to return that argument. 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*.

In functions that accept two arguments *eof-error* and *eof-value*, end of file is an error if *eof-error* is non-nil or if it is unsupplied. If *eof-error* is nil, then the function returns *eof-value* at end of file.

An error is always signaled if end of file is encountered in the middle of an object; for example, if a file does not contain enough right parentheses to balance the left parentheses in it. Mere whitespace does not count as starting an object. If a file contains a symbol or a number immediately followed by end-of-file, it can be read normally without error; if an attempt is made to read further, end of file is encountered immediately and the *eof-* argument(s) obeyed.

These end-of-file conventions are not completely compatible with Maclisp. Maclisp's deviations from this are generally considered to be bugs rather than features.

For Maclisp compatibility, **nil** as the *stream* argument also means to use the value of ***standard-input***, and **t** as the *stream* argument means to use the value of ***terminal-io***. This is only advertised to work in functions that Maclisp has, and should not be written in new programs. Instead of the variable ***standard-input*** Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

The functions below that take *stream* and *eof-option* arguments can also be called with the stream and eof-option in the other order. This functionality is only for compatibility with old Maclisp programs, and should never be used in new programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an **si:io-stream-p** property whose value is **t**.

read &optional *stream eof-option rubout-handler-options*

Reads the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. *rubout-handler-options* are used as options for the rubout handler, if *stream* supports one; see section 22.5, page 500 for more information on this.

cli:read &optional *stream* (*eof-errorp* **t**) *eof-value recursive-p*

The Common Lisp version of **read** differs only in how its arguments are passed.

recursive-p should be non-**nil** when calling from the reader or from the defining function of a read-macro character; that is, when reading a subexpression as part of the task of reading a larger expression. This has two effects: the subexpression is allowed to share ***#n#*** labels with the containing expression, and whitespace which terminates the subexpression (if it is a symbol or number) is not discarded.

read-or-end &optional *stream eof-option rubout-handler-options*

Like **read**, but on an interactive stream if the input is just the character **End** it returns the two values **nil** and **:end**.

read-preserve-delimiters *Variable*

Certain printed representations given to **read**, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close-parenthesis serves to mark the end of the list.) Normally **read** throws away the delimiting character if it is whitespace, but preserves it (using the **:untyi** stream operation) if the character is syntactically meaningful, since it may be the start of the next expression.

If **read-preserve-delimiters** is bound to **t** around a call to **read**, the delimiting character is never thrown away, even if it is whitespace. This may be useful for certain reader macros or special syntaxes.

read-preserving-whitespace &optional *stream* (*eof-errorp* t) *eof-value recursive-p*

Like cli:read but binds read-preserve-delimiters to t. This is the Common Lisp way of requesting the read-preserve-delimiters feature.

read-delimited-list *char* &optional *stream recursive-p*

Reads expressions from *stream* until the character *char* is seen at top level when an expression is expected; then returns a list of the objects read. *char* may be a fixnum or a character object. For example, if *char* is #/], and the text to be read from *stream* is a (b c)] ... then the objects a and (b c) are read, the] is seen as a terminator and discarded, and the value returned is (a (b c)). *recursive-p* is as for cli:read. End of file within this function is always an error since it is always "within an object"—the object whose textual representation is terminated by *char*.

Note that use of this function does not cause *char* to terminate tokens. Usually you want that to happen, but it is purely under the control of the readtable. So you must modify the readtable to make this so. The usual way is to define *char* as a macro character whose defining function just signals an error. The defining function is not called when *char* is encountered in the expected context by read-delimited-list; if *char* is encountered anywhere else, it is an unbalanced bracket and an error is appropriate.

read-for-top-level &optional *stream eof-option*

This is a slightly different version of read. It differs from 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 read would). This version of read is used in the system's "read-eval-print" loops.

read-check-indentation &optional *stream eof-option*

This is like read, but validates the input based on indentation. It assumes that the input data is formatted to follow the usual convention for source files, that an open-parenthesis in column zero indicates a top-level list (with certain specific exceptions). An open-parenthesis in column zero encountered in the middle of a list is more likely to result from close-parentheses missing before it than from a mistake in indentation.

If read-check-indentation finds an open-parenthesis following a return character in the middle of a list, it invents enough close-parentheses to close off all pending lists, and returns. The offending open-parenthesis is :untyi'd so it can begin the next list, as it probably should. End of file in the middle of a list is handled likewise.

read-check-indentation notifies the caller of the incorrect formatting by signaling the condition sys:missing-closeparen. This is how the compiler is able to record a warning about the missing parentheses. If a condition handler proceeds, read goes ahead and invents close-parentheses.

There are a few special forms that are customarily used around function definitions—for example, eval-when, local-declare, and comment. Since it is desirable to begin the function definitions in column zero anyway, read-check-indentation allows a list to begin in column zero within one of these special forms. A non-nil si:may-surround-defun property identifies the symbols for which this is allowed.

read-check-indentation *Variable*

This variable is non-nil during a read in which indentation is being checked.

23.5.1 Non-Stream Parsing Functions

The following functions do expression input but get the characters from a string or a list instead of a stream.

read-from-string *string* &optional *eof-option* (*start* 0) *end*

The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect. If *string* has a fill-pointer it controls how much can be read.

eof-option is what to return if the end of the string is reached, as in **read**. *start* is the index in the string of the first character to be read. *end* is the index at which to stop reading; that point is treated as end of file.

read-from-string returns two values; the first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this is the length of the string.

Example:

```
(read-from-string "(a b c)") => (a b c) and 7
```

cli:read-from-string *string* &optional (*eof-errorp* t) *eof-value* &key (*start* 0) *end* *preserve-whitespace*

The Common Lisp version of **read-from-string** uses a different calling convention. The arguments mean the same thing but are arranged differently. There are three arguments with no counterparts: *eof-errorp* and *eof-value*, which are simply passed on to **cli:read**, and *preserve-whitespace*, which if non-nil means that the reading is done with **read-preserve-delimiters** bound to **t**.

See also the **with-input-from-string** special form (page 473).

parse-integer *string* &key (*start* 0) *end* (*radix* 10.) *junk-allowed*

Parses the contents of *string* (or the portion from *start* to *end*) as a numeral for an integer using the specified radix, and returns the integer. Radices larger than ten are allowed, and they use letters as digits beyond 9. Leading whitespace is always allowed and ignored. A leading sign is also allowed and considered part of the number.

When *junk-allowed* is **nil**, the entire specified portion of string must consist of an integer and leading and trailing whitespace. Otherwise, an error happens.

If *junk-allowed* is non-nil, parsing just stops when a non-digit is encountered. The number parsed so far is returned as the first value, and the index in *string* at which parsing stopped is returned as the second value. This number equals *end* (or the length of *string*) if there is nothing but a number. If non-digits are found without finding a number first, the first value is **nil**. Examples:

```
(parse-integer " 1A " :radix 16.) => 26.
(parse-integer " 15X " :end 3) => 15.
(parse-integer " -15X " :junk-allowed t) => -15. 3
(parse-integer " 15X ") => error!
```

readlist *char-list*

This function is provided mainly for Maclisp compatibility. *char-list* is a list of characters. The characters may be represented by anything that the function **character** accepts: character objects, fixnums, strings, or symbols. The characters are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect.

If there are more characters in *char-list* beyond those needed to define an object, the extra characters are ignored. If there are not enough characters, some kind of **sys:read-end-of-file** error is signaled.

23.5.2 Input Error Conditions

sys:read-error (sys:parse-error error) *Condition*

This condition name classifies all errors detected by the reader per se. Since **sys:parse-error** is implied, all **sys:read-error** errors must provide the proceed type **:no-action** so that automatic proceed is possible if the error happens during compilation. See page 505.

Since this condition name implies **sys:parse-error** and **error**, those two are not mentioned as implications below when **sys:read-error** is.

sys:read-end-of-file (sys:read-error sys:end-of-file) *Condition*

Whenever the reader signals an error for end of file, the condition object possesses this condition name.

Since **sys:end-of-file** is implied, the **:stream** operation on the condition instance returns the stream on which end of file was reached.

sys:read-list-end-of-file *Condition*
(sys:read-end-of-file sys:read-error sys:end-of-file)

This condition is signaled when **read** detects end of file in the middle of a list.

In addition to the **:stream** operation provided because **sys:end-of-file** is one of the proceed types, the condition instance supports the **:list** operation, which returns the list read so far.

Proceed type **:no-action** is provided. If it is used, the reader invents a close-parenthesis to close off the list. Within **read-check-indentation**, the reader signals the error only once, no matter how many levels of list are unterminated.

sys:read-string-end-of-file *Condition*
(sys:read-end-of-file sys:read-error sys:end-of-file)

This is signaled when read detects end of file in the middle of a string delimited by double-quotes.

The :string operation on the condition instance returns the string read so far.

Proceed type :no-action terminates the string and returns. If the string is within other constructs that are unterminated, another end of file error is will be signaled later.

sys:read-symbol-end-of-file *Condition*
(sys:read-end-of-file sys:read-error sys:end-of-file)

This is signaled when read detects end of file within a multiple escape construct.

The :string operation on the condition instance returns the print name read so far.

Proceed type :no-action terminates the symbol and returns. If the symbol is within other constructs that are unterminated, another end of file error is will be signaled later.

sys:missing-closeparen (condition) *Condition*

This condition, which is not an error, is signaled when read-check-indentation finds an open-parenthesis in column zero within a list.

Proceed type :no-action is provided. On proceeding, the reader invents enough close-parentheses to close off all the lists that are pending.

23.6 The Readtable

The syntax used by the reader is controlled by a data structure called the *readtable*. (Some aspects of printing are also controlled by the readtable.) There can be many readtables, but the one that is used is the one which is the value of *readtable*. A particular syntax can be selected for use by setting or binding *readtable* to a readtable which specifies that syntax before reading or printing. In particular, this is how Common Lisp or traditional syntax is selected. The readtable also controls the symbol substitutions which implement the distinction between the traditional and Common Lisp versions of functions such as subst, memberand defstruct.

The functions in this section allow you to modify the syntax of individual characters in a readtable in limited ways. You can also copy a readtable; then you can modify one copy and leave the other unchanged.

A readtables may have one or more names. Named readtables are recorded in a central data base so that you can find a readtable by name. When you copy a readtable, the new one is anonymous and is not recorded in the data base.

readtable *Variable*
readtable *Variable*

The value of readtable or *readtable* is the current readtable. This starts out as the initial standard readtable. You can bind this variable to change temporarily the readtable being used.

The two names are synonymous.

si:standard-readtable *Constant*

This is copied into *readtable* every time the machine is booted. Therefore, it is normally the same as *readtable* unless you make *readtable* be some other readtable. If you alter the contents of *readtable* without setting or binding it to some other readtable, this readtable is changed.

si:initial-readtable *Constant*

The value of si:initial-readtable is a read-only copy of the default current readtable. Its purpose is to preserve a copy of the standard read syntax in case you modify the contents of *readtable* and regret it later. You could use si:initial-readtable as the *from-readtable* argument to copy-readtable or set-syntax-from-char to restore all or part of the standard syntax.

si:common-lisp-readtable *Constant*

A readtable which initially is set up to define Common Lisp read syntax. Reading of Common Lisp programs is done using this readtable.

si:initial-common-lisp-readtable *Constant*

A read-only copy of si:common-lisp-readtable, whose purpose is to preserve a copy of the standard Common Lisp syntax in case you modify si:common-lisp-readtable (such as, by reading a Common Lisp program which modifies the current readtable).

si:rdtbl-names *readtable*

Returns the list of names of readtable. You may setf this to add or remove names.

si:find-readtable-named *name*

Returns the readtable named name, or nil if none is recorded.

readtablep *object*

t if *object* is a readtable.

The user 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 which is called when the character is read. The user can create a completely new readtable, using the readtable compiler (SYS: IO; RTC LISP) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.

copy-readtable &optional *from-readtable to-readtable*

from-readtable, which defaults to the current readtable, is copied. If *from-readtable* is nil, the standard Common Lisp syntax is copied. If *to-readtable* is unsupplied or nil, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copied syntax.

Use copy-readtable to get a private readtable before using the following functions to change the syntax of characters in it. The value of *readtable* at the start of a Lisp Machine session is the initial standard readtable, which usually should not be modified.

set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable*

Copies the syntax of *from-char* in *from-readtable* to character *to-char* in *to-readtable*. *to-readtable* defaults to the current readtable and *from-readtable* defaults to si:initial-standard-readtable (standard traditional syntax).

cli:set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable*

Is a Common Lisp function which copies the syntax of *from-char* in *from-readtable* to character *to-char* in *to-readtable*. *to-readtable* defaults to the current readtable and *from-readtable* defaults to si:initial-common-lisp-readtable (standard Common Lisp syntax).

Common Lisp has a peculiar idea of what it means to copy the syntax of a character. The only aspect of syntax that the readtable supposedly specifies is the choice among

* token constituent: digits, letters, random things like @, !, $, and also colon!
* whitespace: spaces, Tab, Return.
* single escape character: / traditionally, \ in Common Lisp.
* multiple escape character: vertical-bar.
* macro character: standardly ()",.'';
* nonterminating macro character: # is the only such character standardly defined.

The differences among macro characters are determined entirely by the functions that they invoke. The differences among token constituents (including the difference between A and colon) are fixed! You can make A be a macro character, or whitespace, or a quote character, but if you make it a token constituent then it always behaves the way it normally does. You can make colon be a macro character, or whitespace, etc., but if it is a token constituent it always delimits package names. If you make open-parenthesis into a token constituent, there is only one kind of token constituent it can be (it forces the token to be a symbol, like $ or @ or %).

This is not how Lisp Machine readtables really work, but since cli:set-syntax-from-char is provided just for Common Lisp, the behavior specified by Common Lisp is laboriously provided. So, if *from-char* is some kind of token constituent, this function makes *to-char* into a token constituent of the kind that *to-char* is supposed to be—not the kind of token constituent that *from-char* is.

By contrast, the non-Common-Lisp set-syntax-from-char would make *to-char* have exactly the same syntactic properties that *from-char* has.

set-character-translation *from-char to-char* &optional *readtable*

Changes *readtable* so that *from-char* will be translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lower case letters to upper case. Character translation is inhibited by escape characters and within strings. *readtable* defaults to the current readtable.

The following syntax-setting functions are more or less obsolete.

set-syntax-from-description *char description* &optional *readtable*

Sets the syntax of *char* in *readtable* to be that described by the symbol *description*. *readtable* defaults to the current readtable.

Each readtable has its own set of descriptions which it defines. The following descriptions are defined in the standard readtable:

si:alphabetic	An ordinary character such as 'A'.
si:break	A token separator such as '('. (Obviously left parenthesis has other properties besides being a break.
si:whitespace	A token separator that can be ignored, such as ' '.
si:single	A self-delimiting single-character symbol. The initial readtable does not contain any of these.
si:escape	The character quoter. In the initial readtable this is '/'.
si:multiple-escape	The symbol print-name quoter. In the initial readtable this is '|'.
si:macro	A macro character. Don't use this; use set-macro-character (page 540).
si:non-terminating-macro	A macro character recognized only at the start of a token. In the initial readtable, '#' is such a character. (It is also a dispatching macro, but that is another matter.) The correct way to make a character be a macro is with set-macro-character.
si:character-code-escape	The octal escape for special characters. In the initial readtable this is '⊗'.
si:digitscale	a character for shifting an integer by digits. In the initial readtable this is '^'.
si:bitscale	A character for shifting an integer by bits. In the initial readtable this is '_' (underscore).
si:slash si:circlecross	Obsolete synonyms for si:escape and si:character-code-escape.

Unfortunately it is no longer possible to provide si:doublequote as double-quote is now an ordinary macro character.

These symbols may be moved to the keyword package at some point.

setsyntax *character arg2 arg3*

This exists only for Maclisp compatibility. The above functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be a fixnum, a symbol, or a string, i.e. anything acceptable to the character function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro	The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, may tyi or read from *standard-input* (i.e. may call tyi or read without specifying a stream), and returns an object which is taken as the result of the read.
:splicing	Like :macro but the object returned by the macro function is a list that is nconced into the list being read. If the character is read anywhere except inside a list (at top level or after a dotted-pair dot), then it may return (), which means it is ignored, or (*obj*), which means that *obj* is read.
:single	The character becomes a self-delimiting single-character symbol. If *arg3* is a fixnum, the character is translated to that character.
nil	The syntax of the character is not changed, but if *arg3* is a fixnum, the character is translated to that character.
a symbol	The syntax of the character is changed to be the same as that of the character *arg2* in the standard initial readtable. *arg2* is converted to a character by taking the first character of its print name. Also if *arg3* is a fixnum, the character is translated to that character.

23.7 Read-Macro Characters

A *read-macro character* (or just *macro character*) is a character whose syntax is defined by a function which the reader calls whenever that character is seen (unless it is escaped). This function can optionally read additional characters and is then responsible for returning the object which they represent.

The standard meanings of the characters open-parenthesis, semicolon, single-quote, double-quote, #, backquote (`) and comma are implemented by making them macro characters.

For example, open-parenthesis is implemented as a macro character whose defining function reads expressions until a close-parenthesis is found, throws away the close-parenthesis, and returns a list of the expressions read. (It actually must be more complicated than this in order to deal properly with dotted lists and with indentation checking.) Semicolon is implemented as a macro character whose defining function swallows characters until a Return and then returns no values.

Close-parenthesis and close-horseshoe (⊃) are also macro characters so that they will terminate symbols. Their defining functions signal errors if actually called; but when these delimiters are encountered in their legitimate contexts they are recognized and handled specially before the defining function is called.

The user can also define macro characters.

When a macro's defining function is called, it receives two arguments: the input stream, and the macro character being handled. The function may read characters from the stream, and should return zero or more values, which are the objects that the macro construct "reads as". Zero values causes the macro construct to be ignored (the semicolon macro character does this), and one value causes the macro construct to read as a single object (most macro characters do this). More than one value is allowed only within a list.

Macro characters may be *terminating* or *non-terminating*. A non-terminating macro character is only recognized as a macro character when it appears at the beginning of a token. If it appears when a token is already in progress, it is treated as a symbol constituent. Of the standard macro characters, all but # are terminating.

One kind of macro character is the *dispatch* macro character. This kind of character is handled by reading one more character, converting it to upper case, and looking it up in a table. Thus, the dispatch macro character is the start of a two-character sequence, with which is associated a defining function. # is the only standardly defined dispatch macro character.

When a dispatch macro character is used, it may be followed by a decimal integer which serves as a parameter. The character for the dispatch is actually the first non-digit seen.

The defining function for a dispatch macro two-character sequence is almost like that of an ordinary macro character. However, it receives one more argument. This is the parameter, the decimal integer that followed the dispatch macro character, or **nil** if no parameter was written. Also, the second argument is the subdispatch character, the second character of the sequence. The dispatch macro character itself is not available.

set-macro-character *char function* &optional *non-terminating-p in-readtable*

Sets the syntax of character *char* in readtable *in-readtable* to be that of a macro character which is handled by *function*. When that character is read by **read**, *function* is called.

char is made a non-terminating macro character if *non-terminating-p* is non-**nil**, a terminating one otherwise.

get-macro-character *char in-readtable*

Returns two values that describe the macro character status of *char* in *in-readtable*. If *char* is not a macro character, both values are **nil**. Otherwise, the first value is the *function* and the second value is the *non-terminating-p* for this character.

Those two values, passed to **set-macro-character**, are usually sufficient to recreate exactly the syntax *char* has now; however, since one of the arguments that the function receives is the macro character that invoked it, it may not behave the same if installed on a different character or in a different readtable. In particular, the definition of a dispatch macro character is a standard function that looks the macro character up in the readtable. Thus, the definition only records that the macro character *is* a dispatch macro character; it does not say what subcharacters are allowed or what they mean.

make-dispatch-macro-character *char* &optional *non-terminating-p in-readtable*
Makes *char* be a dispatch macro character in *in-readtable*. This means that when *char* is seen read will read one more character to decide what to do. # is an example of a dispatch macro character. *non-terminating-p* means the same thing as in set-macro-character.

set-dispatch-macro-character *char subchar function* &optional *in-readtable*
Sets the syntax of the two-character sequence *char subchar*, assuming that *char* is already a dispatch macro character. *function* becomes the defining function for this sequence.

If *subchar* is lower case, it is converted to upper case. Case is never significant for the character that follows a dispatch macro character. The decimal digits may not be defined as subchars since they are always used for infix numeric arguments as in #5r.

get-dispatch-macro-character *char subchar* &optional *in-readtable*
Returns the *function* for *subchar* following dispatch macro character *char* in readtable *in-readtable*. The value is nil if *subchar* is not defined for following *char*.

These subroutines are for use by the defining functions of macro characters. Ordinary read should not be used for reading subexpressions, and the ordinary :tyi operation or functions read-char or tyi should not be used for single-character input. The functions below should be used instead.

si:read-recursive &optional *stream*
Equivalent to (cli:read *stream* t nil t). See page 531. This is the recommended way for a macro character's defining function to read a subexpression.

si:xr-xrtyi *stream ignore-whitespace no-chars-special no-multiple-escapes*
Reads the next input character from *stream*, for a macro character's defining function. If *ignore-whitespace* is non-nil, any whitespace characters seen are discarded and the first non-whitespace character is returned.

The first value is the character as translated; the third value is the original character, before translation. The second value is a syntax code which is of no interest to users except to be passed to si:xr-xruntyi if this character must be unread.

Normally, this function processes all the escape characters, and performs translations (such as from lower case letters to upper case letters) on characters not escaped. Font specifiers (epsilons followed by digits or *) are ignored if the file is formatted using them.

If *no-multiple-escapes* is non-nil, multiple escapes (vertical bar characters) are not processed; they are returned to the caller. This mode is used for reading the contents of strings. If *no-chars-special* is non-nil, no escape characters are processed. All characters are simply returned to the caller (except that font specifiers are still discarded if appropriate).

si:xr-xruntyi *stream char num*

Unreads *char*, for a macro character's defining function. *char* should be the third value returned by the last call to si:xr-xrtyi, and *num* should be the second value.

read-suppress *Variable*

If this variable is non-nil, all the standard read functions and macro characters do their best to avoid any errors, and any side effects except for removing characters from the input stream. For example, symbols are not interned to avoid either errors (for nonexistent packages) or side effects (adding new symbols to packages). In fact, nil is used in place of any symbol that is written.

User macro characters should also notice this variable when appropriate.

The purpose of the variable is to allow expressions to be skipped and discarded. The read-time conditional constructs #+ and #- bind it to t to skip the following expression if it is not wanted.

The following functions for defining macro characters are more or less obsolete.

set-syntax-macro-char *char function* &optional *readtable*

Changes *readtable* so that *char* is a macro character. When *char* is read, *function* is called. *readtable* defaults to the current readtable.

function is called with two arguments, *list-so-far* and the input stream. When a list is being read, *list-so-far* is that list (nil if this is the first element). At the top level of read, *list-so-far* is the symbol :toplevel. After a dotted-pair dot, *list-so-far* is the symbol :after-dot. *function* may read any number of characters from the input stream and process them however it likes.

function should return three values, called *thing*, *type*, and *splice-p*. *thing* is the object read. If *splice-p* is nil, *thing* is the result. If *splice-p* is non-nil, then when reading a list *thing* replaces the list being read—often it will be *list-so-far* with something else nconc'ed onto the end. At top-level and after a dot, if *splice-p* is non-nil the *thing* is ignored and the macro-character does not contribute anything to the result of read. *type* is a historical artifact and is not really used; nil is a safe value. Most macro character functions return just one value and let the other two default to nil.

Note that the convention for values returned by *function* is different from that used for functions specified in set-macro-character, above. set-syntax-macro-char works by encapsulating *function* in a closure to convert the values to the sort that set-macro-character wants and then passing the closure to set-macro-character.

function should not have any side-effects other than on the stream and *list-so-far*. Because of the way the rubout-handler works, *function* can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called :macro syntax).

set-syntax-#-macro-char *char function* &optional *readtable*

Causes *function* to be called when *#char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters, documented above. When *function* is called, the special variable si:xr-sharp-argument contains nil or a number that is the number or special bits between the # and *char*.

setsyntax-sharp-macro *character type function* &optional *readtable*

This exists only for Maclisp compatibility. set-dispatch-macro-character should be used instead. If *function* is nil, *#character* is turned off, otherwise it becomes a macro that calls *function*. *type* can be :macro, :peek-macro, :splicing, or :peek-splicing. The splicing part controls whether *function* returns a single object or a list of objects. Specifying peek causes *character* to remain in the input stream when *function* is called; this is useful if *character* is something like a left parenthesis. *function* gets one argument, which is nil or the number between the # and the *character*.

23.8 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 most streams do not support them.

If the 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, read's *eof-option* if it had one or a magic internal marker if it didn't. Whatever the stream returns is what read returns. If the stream wants to implement the :read operation by internally calling 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 may 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 *depth* (for comparison against the *print-level* variable), and *escape-p* (which is the value of *print-escape*). 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 may be changed without much notice.

si:print-object *object depth stream* &optional *which-operations*

Outputs the printed-representation of *object* to *stream*, as modified by *depth* and the values of the *print-... variables.

This is the internal guts of the Lisp printer. When a stream's :print handler calls this function, it should supply the list (:string-out) for *which-operations*, to prevent itself from being called recursively. Or it can supply nil if it does not want to receive :string-out messages.

If you want to customize the behavior of all printing of Lisp objects, advising (see section 30.10, page 742) this function is the way to do it. See section 23.1, page 513.

si:print-list *list depth stream which-operations*
This is the part of the Lisp printer that prints lists. A stream's :print handler can call this function, passing along its own arguments and its own which-operations, to arrange for a list to be printed the normal way and the stream's :print hook to get a chance at each of the list's elements.

24. Naming of Files

A Lisp Machine generally has access to many file systems. While it may have its own file system on its own disks, usually a community of Lisp Machine users want to have a shared file system accessible by any of the Lisp Machines over a network. These shared file systems can be implemented by any computer that is capable of providing file system service. A file server computer may be a special-purpose computer that does nothing but service file system requests from computers on a network, or it may be a time-sharing system.

Programs need to use names to designate files within these file systems. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, in the ITS file system, a typical name looks like:

```
                    DSK: GEORGE; FOO QFASL
```

with DSK being a device name, GEORGE being a directory name, FOO being the first file name and QFASL being the second file name. However, in TOPS-20, a similar file name is expressed as:

```
                    PS:<GEORGE>FOO.QFASL
```

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists, or new formats that could get added in the future. However, existing programs should retain their abilities to manipulate the names.

The functions and flavors described in this chapter exist to solve this problem. They provide an interface through which a program can deal with names of files and manipulate them without depending on anything about their syntax. This lets a program deal with multiple remote file servers simultaneously, using a uniform set of conventions.

24.1 Pathnames

All file systems dealt with by the Lisp Machine are mapped into a common model, in which files are named by something called a *pathname*. A pathname always has six components, each with a standard meaning. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. Pathname components are described in the following section, and the mappings between components and user syntax is described for each file system later in this chapter.

pathnamep *object*

t if *object* is a pathname.

A pathname is an instance of a flavor (see chapter 21, page 401); exactly which flavor depends on what the host of the pathname is, but **pathname** is always one of its component flavors. If *p* is a pathname, then **(typep *p* 'pathname)** returns **t**. One of the messages handled by host objects is the **:pathname-flavor** operation, which returns the name of the flavor to use for pathnames on that host. And one of the differences between host flavors is how they handle this operation.

There are functions for manipulating pathnames, and there are also messages that can be sent to them. These are described later in this chapter.

Two important operations of the pathname system are *parsing* and *merging*. Parsing is the conversion of a string—which might be something typed in by the user when asked to supply the name of a file—into a pathname object. This involves finding out what host the pathname is for, then using the file name syntax conventions of that host to parse the string into the standard pathname components. Merging is the operation that takes a pathname with missing components and supplies values for those components from a set of defaults.

The function **string**, applied to a pathname, converts it into a string that is in the file name syntax of its host's file system, except that the name of the host followed by a colon is inserted at the front. This is the inverse of parsing. **princ** of a pathname also does this, then prints the contents of the string. Flavor operations such as **:string-for-dired** exist which convert all or part of a pathname to a string in other fashions that are designed for specific applications. **prin1** of a pathname prints the pathname using the **#c** syntax so it can be read back in to produce an equivalent pathname (or the same pathname, if read in the same session).

Since each kind of file server can have its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and figuring out what each component is. The parsers all work differently. How can the parsing operation know which parser to use? The first thing that the parser does is to figure out which host this filename belongs to. A filename character string may specify a host explicitly by having the name of the host, followed by a colon, at either the beginning or the end of the string. For example, the following strings all specify hosts explicitly:

```
AI: COMMON; GEE WHIZ              ; This specifies host AI.
COMMON; GEE WHIZ AI:              ; So does this.
AI: ARC: USERS1; FOO BAR          ; So does this.
ARC: USERS1; FOO BAR AI:          ; So does this.
EE:PS:<COMMON>GEE.WHIZ.5          ; This specifies host EE.
PS:<COMMON>GEE.WHIZ.5 EE:         ; So does this.
```

If the string does not specify a host explicitly, the parser chooses a host by default and uses the syntax for that host. The optional arguments passed to the parsing function (**fs:parse-pathname**) tell it which host to assume. Note: the parser is not confused by strings starting with **DSK:** or **PS:** because it knows that neither of those is a valid host name. But if the default host has a device whose name happens to match the name of some host, you can prevent the device name from being misinterpreted as a host name by writing an extra colon at the beginning of the string: For example, **:EE:<RMS>FOO.BAR** refers to the device **EE** on the default host (assumed to use TOPS-20 syntax) rather than to the host named **EE**.

Pathnames are kept unique, like symbols, so that there is only one object with a given set of components. This is useful because a pathname object has a property list (see section 5.10, page 113) on which you can store properties describing the file or family of files that the pathname represents. The uniqueness implies that each time the same components are typed in, the program gets the same pathname object and finds there the properties it ought to find.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; a pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with **:newest** as its version refers to the same file as a pathname which has the appropriate number as the version. In systems with links, multiple file names, logical devices, etc., two pathnames that look quite different may really turn out to address the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

When you want to store properties describing an individual file, use the pathname you get by sending **:truename** to a stream rather than the pathname you open. This avoids problems with different pathnames that refer to the same file.

To get a unique pathname object representing a family of files, send the message **:generic-pathname** to a pathname for any file in the family (see section 24.5, page 561).

24.2 Pathname Components

These are the components of a pathname. They are clarified by an example below.

host	An object that represents the file system machine on which the file resides. A host object is an instance of a flavor one of whose components is **si:basic-host**. The precise flavor varies depending on the type of file system and how the files are to be accessed.
device	Corresponds to the "device" or "file structure" concept in many host file systems.
directory	The name of a group of related files belonging to a single user or project. Corresponds to the "directory" concept in many host file systems.
name	The name of a group of files that can be thought of as conceptually the "same" file. Many host file systems have a concept of "name" which maps directly into this component.
type	Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is; such as, a Lisp source file, a QFASL file, etc.
version	Corresponds to the "version number" concept in many host file systems. This is a number that increments every time the file is modified. Some host systems do not support version numbers.

As an example, consider a Lisp program named **CONCH**. If it belongs to **GEORGE**, who uses the **FISH** machine, the host would be the host-object for the machine **FISH**, the device would probably be the default and the directory would be **GEORGE**. On this directory would be a number of files related to the **CONCH** program. The source code for this program would live in a set of files with name **CONCH**, type **LISP**, and versions **1**, **2**, **3**, etc. The compiled form of the program would live in files named **CONCH** with type **QFASL**; each would have the same version number as the source file that it came from. If the program had a documentation file, it would have type **INFO**.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is **nil**. Before a file server can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in from defaults. But pathnames with missing components are often handed around inside the machine, since almost all pathnames typed by users do not specify all the components explicitly. The host is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to know what its host is. All pathnames have host attributes, even if the string being parsed does not specify one explicitly.

A component of a pathname can also be the special symbol :unspecific. :unspecific means, explicitly, "this component has been specified as missing", whereas nil means that the component was not specified and should default. In merging, :unspecific counts as a specified component and is not replaced by a default. :unspecific does *not* mean "unspecified"; it is unfortunate that those two words are similar.

:unspecific is used in *generic* pathnames, which refer not to a file but to a whole family of files. The version, and usually the type, of a generic pathname are :unspecific. Another way :unspecific is used has to do with mapping of pathnames into file systems such as ITS that do not have all six components. A component that is really "not there" is :unspecific in the pathname. When a pathname is converted to a string, nil and :unspecific both cause the component not to appear in the string.

A component of a pathname can also be the special symbol :wild. This is useful only when the pathname is being used with a directory primitive such as fs:directory-list (see page 598), where it means that this pathname component matches anything. The printed representation of a pathname usually designates :wild with an asterisk; however, this is host-dependent.

What values are allowed for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually just treated as something supplied by the user that only needs to be remembered.

In general, programs can interpret the components of a pathname independent of the file system; and a certain minimum set of possible values of each component are supported on all file systems. The same pathname component value may have very different representations when the pathname is made into a string, depending on the file system. This does not affect programs that operate on the components. The user, when asked to type a pathname, always uses the system-dependent string representation. This is convenient for the user who moves between using the Lisp Machine on files stored on another host and making direct use of that host. However, when the mapping between string form and components is complicated, the components may not be obvious from what you type.

The type is always a string, or one of the special symbols nil, :unspecific, and :wild. Certain hosts impose a limit on the size of string allowed, often very small. Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are usually "LISP", compiled Lisp programs are "QFASL", etc. However, these file type conventions are host-specific, for the important reason that some hosts do not allow a string five characters long to be used as the type. Therefore, programs should use a *canonical type* rather than an actual string to specify their conventional default file types. Canonical types are described below.

For the version, it is always legitimate to use a positive fixnum, or certain special symbols. **nil**, **:unspecific**, and **:wild** have been explained above. The other standardly allowed symbols are **:newest** and **:oldest**. **:newest** refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. **:oldest** refers to the smallest version number that exists. Some file systems may define other special version symbols, such as **:installed** for example, or may allow negative numbers. Some do not support versions at all. Then a pathname may still contain any of the standard version components, but it does not matter what the value is.

The device, directory, and name are more system-dependent. These can be strings (with host-dependent rules on allowed characters and length) or they can be *structured*. A structured component is a list of strings. This is used for file system features such as hierarchical directories. The system is arranged so that programs do not need to know about structured components unless they do host-dependent operations. Giving a string as a pathname component to a host that wants a structured value converts the string to the appropriate form. Giving a structured component to a host that does not understand them converts it to a string by taking the first element and ignoring the rest.

Some host file systems have features that do not fit into this pathname model. For instance, directories might be accessible as files, there might be complicated structure in the directories or names, or there might be relative directories, such as '<' in Multics. These features appear in the parsing of strings into pathnames, which is one reason why the strings are written in host-dependent syntax. Pathnames for hosts with these features are also likely to handle additional messages besides the common ones documented in this chapter, for the benefit of host-dependent programs that want to access those features. However, once your program depends on any such features, it will work only for certain file servers and not others; in general, it is a good idea to make your program work just as well no matter what file server is being used.

24.2.1 Raw Components and Interchange Components

On some host file systems it is conventional to use lower-case letters in file names, while in others upper case is customary, or possibly required. When pathname components are moved from pathnames of one file system to pathnames of another file system, it is useful to convert the case if necessary so that you get the right case convention for the latter file system as a default. This is especially useful when copying files from one file system to another.

The Lisp Machine system defines two representations for each of several pathname components (the device, directory, name and type). There is the *raw* form, which is what actually appears in the filename on the host file system, and there is the *interchange* form, which may differ in alphabetic case from the raw form. The raw form is what is stored inside the pathname object itself, but programs nearly always operate on the interchange form. The **:name**, **:type**, etc., operations return the interchange form, and the **:new-name**, etc., operations expect the interchange form. Additional operations **:raw-name**, etc., are provided for working with the raw components, but these are rarely needed.

The interchange form is defined so that it is always customarily in upper case. If upper case is customary on the host file system, then the interchange form of a component is the same as the raw form. If lower case is customary on the host file system, as on Unix, then the interchange form has case inverted. More precisely, an all-upper-case component is changed to all-lower-case, an all-lower-case component is changed to all-upper-case, and a mixed-case component is not changed. (This is a one-to-one mapping). Thus, a Unix pathname with a name component of "foo" has an interchange-format name of "FOO", and vice versa.

For host file systems which record case when files are created but ignore case when comparing filenames, the interchange form is always upper case.

The host component is not really a name, and case is always ignored in host names, so there is no need for two forms of host component. The version component does not need them either, because it is never a string.

24.2.2 Pathname Component Operations

:host *Operation on* pathname
:device *Operation on* pathname
:directory *Operation on* pathname
:name *Operation on* pathname
:type *Operation on* pathname
:version *Operation on* pathname

These return the components of the pathname, in interchange form. The returned values can be strings, special symbols, or lists of strings in the case of structured components. The type is always a string or a symbol. The version is always a number or a symbol.

:raw-device *Operation on* pathname
:raw-directory *Operation on* pathname
:raw-name *Operation on* pathname
:raw-type *Operation on* pathname

These return the components of the pathname, in raw form.

:new-device *dev* *Operation on* pathname
:new-directory *dir* *Operation on* pathname
:new-name *name* *Operation on* pathname
:new-type *type* *Operation on* pathname
:new-version *version* *Operation on* pathname

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as being in interchange form, which means its case may be converted. The :new-device, :new-directory and :new-name operations accept a string (or a special symbol) or a list that is a structured name. If the host does not define structured components, and you specify a list, its first element is used.

:new-raw-device *dev* *Operation on* pathname
:new-raw-directory *dir* *Operation on* pathname
:new-raw-name *name* *Operation on* pathname
:new-raw-type *type* *Operation on* pathname

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as raw.

:new-suggested-name *name* *Operation on* pathname
:new-suggested-directory *dir* *Operation on* pathname

These differ from the :new-name and :new-directory operations in that the new pathname constructed has a name or directory based on the suggestion, but not necessarily identical to it. It tries, in a system-dependent manner, to adapt the suggested name or directory to the usual customs of the file system in use.

For example, on a TOPS-20 system, these operations would convert *name* or *dir* to upper case, because while lower-case letters *may* appear in TOPS-20 pathnames, it is not customary to generate such pathnames by default.

:new-pathname &rest *options* *Operation on* pathname

This returns a new pathname that is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are :host, :device, :directory, :name, :type, and :version. Alternatively, the keywords :raw-device, :raw-directory, :raw-name and :raw-type may be used to specify a component in raw form.

Two additional keywords, :canonical-type and :original-type, allow the type field to be specified as a canonical type. See the following section for a description of canonical types. Also, the value specified for the keyword :type may be a canonical type symbol.

If an invalid component is specified, it is replaced by some valid component so that a valid pathname can be returned. You can tell whether a component is valid by specifying it in :new-pathname and seeing whether that component of the resulting pathname matches what you specified.

The operations :new-name, etc., are equivalent to :new-pathname specifying only one component to be changed; in fact, that is how those operations are implemented.

24.2.3 Canonical Types

Canonical types are a way of specifying a pathname type component using host-dependent conventions without making the program itself explicitly host dependent. For example, the function compile-file normally provides a default type of "LISP", but on VMS systems the default must be "LSP" instead, and on Unix systems it is "l". What compile-file actually does is to use a canonical type, the keyword :lisp, as the default. This keyword is given a definition as a canonical type, which specifies what it maps into on various file systems.

A single canonical type may have more than one mapping on a particular file system. For example, on TOPS-20 systems the canonical type :LISP maps into either "LISP" or "LSP". One of the possibilities is marked as "preferred"; in this case, it is "LISP". The effect of this is that either FOO.LISP or FOO.LSP would be acceptable as having canonical type :lisp, but merging yields "LISP" as the type when defaulting from :lisp.

Note that the canonical type of a pathname is not a distinct component. It is another way of describing or specifying the type component.

A canonical type must be defined before it is used.

fs:define-canonical-type *Macro*
symbol standard-mapping system-dependent-mappings...

Defines *symbol* as a canonical type. *standard-mapping* is the actual type component that it maps into (a string), with exceptions as specified by *system-dependent-mappings*. Each element of *system-dependent-mappings* (that is, each additional argument) is a list of the form

(*system-type preferred-mapping other-mappings*...)

system-type is one of the system-type keywords the :system-type operation on a host object can return, such as :unix, :tops20, and :lispm (see page 577). The argument describes how to map this canonical type on that type of file system. *preferred-map* (a string) is the preferred mapping of the canonical type, and *other-mappings* are additional strings that are accepted as matching the canonical type.

system-type may also be a list of system types. Then the argument applies to all of those types of file systems.

All of the mapping strings are in interchange form.

For example, the canonical type :lisp is defined as follows:

```
(fs:define-canonical-type :lisp "LISP"
  (:unix "L" "LISP")
  (:vms "LSP")
  ((:tops20 :tenex) "LISP" "LSP"))
```

Other canonical types defined by the system include :qfasl, :text, :press, :qwabl, :babyl, :mail, :xmail, :init, :patch-directory, :midas, :palx, :unfasl, :widths, :output, mac, tasm, doc, mss, tex, pl1 and clu. The standard mapping for each is the symbol's pname.

To match a pathname against a canonical type, use the :canonical-type operation.

:canonical-type *Operation on* pathname

Returns two values which describe whether and how this pathname's type component matches any canonical type.

If the type component is one of the possible mappings of some canonical type, the first value is that canonical type (the symbol). The second value is nil if the type component is the preferred mapping of the canonical type; otherwise it is the actual type component, in interchange form. The second value is called the *original type* of the pathname.

If the type component does not match a canonical type, the first value is the type component in interchange form (a string), and the second value is nil.

This operation is useful in matching a pathname against a canonical type; the first value is eq to the canonical type if the pathname matches it. The operation is also useful for transferring a type field from one file system to another while preserving canonical type; this is described below.

A new pathname may also be constructed by specifying a canonical type.

:new-canonical-type *canonical-type* &optional *original-type* — *Operation on* pathname

Returns a pathname different from this one in having a type component that matches *canonical-type*.

If *original-type* is a possible mapping for *canonical-type* on this pathname's host, then it is used as the type component. Otherwise, the preferred mapping for *canonical-type* is used. If *original-type* is not specified, it defaults to this pathname's type component. If it is specified as nil, the preferred mapping of the canonical type is always used. If *canonical-type* is a string rather than an actual canonical type, it is used directly as the type component, and the *original-type* does not matter.

The :new-pathname operation accepts the keywords :canonical-type and :original-type. The :new-canonical-type operation is equivalent to :new-pathname with those keywords.

Suppose you wish to copy the file named *old-pathname* to a directory named *target-directory-pathname*, possibly on another host, while preserving the name, version and canonical type. That is, if the original file has a name acceptable for a QFASL file, the new file should also. Here is how to compute the new pathname:

```
(multiple-value-bind (canonical original)
    (send old-pathname :canonical-type)
  (send target-directory-pathname :new-pathname
        :name (send old-pathname :name)
        :version (send old-pathname :version)
        :canonical-type canonical
        :original-type original))
```

Suppose that *old-pathname* is OZ:<FOO>A.LISP.5, where OZ is a TOPS-20, and the target directory is on a VMS host. Then canonical is :lisp and original is "LISP". Since "LISP" is not an acceptable mapping for :lisp on a VMS system, the resulting pathname has as its type component the preferred mapping for :lisp on VMS, namely, "LSP".

But if the target host is a Unix host, the new file's type is "LISP", since that is an acceptable (though not preferred) mapping for :lisp on Unix hosts. If you would rather that the preferred mapping always be used for the new file's type, omit the :original-type argument to the :new-pathname operation. This would result in a type component of "L" in interchange form, or "l" in raw form, in the new file's pathname.

The function compile-file actually does something cleverer than using the canonical type as a default. Doing that, and opening the resulting pathname, would look only for the preferred mapping of the canonical type. compile-file actually tries to open *each* possible mapping, trying the preferred mapping first. Here is how it does so:

:open-canonical-default-type *Operation on* pathname
canonical-type &rest *options*

If this pathname's type component is non-nil, the pathname is simply opened, passing the *options* to the :open operation. If the type component is nil, each mapping of *canonical-type* is tried as a type component, in the order the mappings appear in the canonical type definition. If an open succeeds, a stream is returned. The possibilities continue to be tried as long as fs:file-not-found errors happen; other errors are not handled. If all the possibilities fail, a fs:file-not-found error is signaled for the caller, with a pathname that contains the preferred mapping as its type component.

24.3 Defaults and Merging

When the user is asked to type in a pathname, it is of course unreasonable to require the user to type a complete pathname, containing all components. Instead there are defaults, so that components not specified by the user can be supplied automatically by the system. Each program that deals with pathnames typically has its own set of defaults.

The system defines an object called a *defaults alist*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults alist can remember more than one default pathname if defaults are being kept separately for each host; this is controlled by the variable fs:*defaults-are-per-host*. The main primitive for using defaults is the function fs:merge-pathname-defaults (see page 558).

In place of a defaults alist, you may use just a pathname. Defaulting one pathname from another is useful for cases such as a program that has an input file and an output file, and asks the user for the name of both, letting the unsupplied components of one name default from the other. Unspecified components of the output pathname come from the input pathname, except that the type should default not to the type of the input but to the appropriate default type for output from this program.

The implementation of a defaults alist is an association list of host names and default pathnames. The host name nil is special and holds the defaults for all hosts, when defaults are not per-host.

The *merging* operation takes as input a pathname, a defaults alist (or another pathname), a default type, and a default version, and returns a pathname. Basically, the missing components in the pathname are filled in from the defaults alist. However, if a name is specified but the type or version is not, then the type or version is treated specially.

Here are the merging rules in full detail.

If no host is specified, the host is taken from the defaults. If the pathname explicitly specifies a host and does not supply a device, then the the default file device for that host is used.

If the pathname specifies a device named DSK, that is replaced with the *working device* for the pathname's host, and the directory defaults to the *working directory* for the host if it is not specified. See fs:set-host-working-directory, below.

Next, if the pathname does not specify a host, device, directory, or name, that component comes from the defaults.

If the value of **fs:*always-merge-type-and-version*** is non-**nil**, the type and version are merged just like the other components.

If **fs:*always-merge-type-and-version*** is **nil**, as it normally is, the merging rules for the type and version are more complicated and depend on whether the pathname specifies a name. If the pathname doesn't specify a name, then the type and version, if not provided, come from the defaults, just like the other components. However, if the pathname does specify a name, then the type and version come from the *default-type* and *default-version* arguments to **merge-pathname-defaults**. If those arguments were omitted, the value of **fs:*name-specified-default-type*** (initially, **:lisp**) is used as the default type, and **:newest** is used as the default version.

The reason for this is that the type and version "belong to" some other filename, and are thought to be unlikely to have anything to do with the new filename you are typing in.

fs:set-host-working-directory *host pathname*

Sets the *working device* and *working directory* for *host* to those specified in *pathname*. *host* should be a host object or the name of a host. *pathname* may be a string or a pathname. The working device and working directory are used for defaulting pathnames in which the device is specified as **DSK**.

The editor command **Meta-X Set Working Directory** provides a convenient interface to this function.

The following special variables are parts of the pathname interface that are relevant to defaults.

fs:*defaults-are-per-host* *Variable*

This is a user customization option intended to be set by a user's **LISPM INIT** file (see section 35.8, page 800). The default value is **nil**, which means that each program's set of defaults contains only one default pathname. If you type in just a host name and a colon, the other components of the name default from the previous host, with appropriate translation to the new host's pathname syntax. If **fs:*defaults-are-per-host*** is set to **t**, each program's set of defaults maintains a separate default pathname for each host. If you type in just a host name and a colon, the last file that was referenced on that host is used.

fs:*always-merge-type-and-version* *Variable*

If this variable is non-**nil**, then the type and version are defaulted only from the pathname defaults just like the other components.

fs:*name-specified-default-type* *Variable*

If **fs:*always-merge-type-and-version*** is **nil**, then when a name is specified but not a type, the type defaults from an argument to the merging function. If that argument is not specified, this variable's value is used. It may be a string or a canonical type keyword. The value is initially **:lisp**.

`*default-pathname-defaults*` *Variable*

This is the default defaults alist; if the pathname primitives that need a set of defaults are not given one, they use this one. Most programs, however, should have their own defaults rather than using these.

`cli:*default-pathname-defaults*` *Variable*

The Common Lisp version of the default pathname defaults. The value of this variable is a pathname rather than an alist. This variable is magically (with a forwarding pointer) identified with a cell in the defaults-alist which the system really uses, so that setting this variable modifies the contents of the alist.

`fs:last-file-opened` *Variable*

This is the pathname of the last file that was opened. Occasionally this is useful as a default. Since some programs deal with files without notifying the user, you must not expect the user to know what the value of this symbol is. Using this symbol as a default may cause unfortunate surprises if you don't announce it first, and so such use is discouraged.

These functions are used to manipulate defaults alists directly.

`fs:make-pathname-defaults`

Creates a defaults alist initially containing no defaults. If you ask this empty set of defaults for its default pathname before anything has been stored into it you get the file FOO on the user's home directory on the host he logged in to.

`fs:copy-pathname-defaults` *defaults*

Creates a defaults alist, initially a copy of *defaults*.

`fs:default-pathname` &optional *defaults host default-type default-version*

This is the primitive function for getting a default pathname out of a defaults alist. Specifying the optional arguments *host*, *default-type*, and *default-version* to be non-nil forces those fields of the returned pathname to contain those values.

If fs:*defaults-are-per-host* is nil (its default value), this gets the one relevant default from the alist. If it is t, this gets the default for *host* if one is specified, otherwise for the host most recently used.

If *defaults* is not specified, the default defaults are used.

This function also has an additional optional argument *internal-p*, which is obsolete.

`fs:default-host` *defaults*

Returns the default host object specified by the defaults-alist *defaults*. This is the host used by pathname defaulting with the given defaults if no host is specified.

fs:set-default-pathname *pathname* &optional *defaults*
This is the primitive function for updating a set of defaults. It stores *pathname* into *defaults*. If *defaults* is not specified, the default defaults are used.

24.4 Pathname Functions

This function obtains a pathname from an object if that is possible.

pathname *object*
Converts *object* to a pathname and returns that, if possible. If *object* is a string or symbol, it is parsed. If *object* is a plausible stream, it is asked for its pathname with the :pathname operation. If *object* is a pathname, it is simply returned. Any other kind of *object* causes an error.

These functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

parse-namestring *thing* &optional *host defaults* &key (*start* 0) *end junk-allowed*
Is the Common Lisp function for parsing file names. It is equivalent to fs:parse-pathname except in that it takes some keyword arguments where the other function takes all positional arguments.

fs:parse-pathname *thing* &optional *host defaults* (*start* 0) *end junk-allowed*
This turns *thing*, which can be a pathname, a string, a symbol, or a Maclisp-style name list, into a pathname. Most functions that are advertised to take a pathname argument call fs:parse-pathname on it so that they can accept anything that can be turned into a pathname. If thing is itself a pathname, it is returned unchanged.

If *thing* is a string, *start* and *end* are interpreted as indices specifying a substring to parse. They are just like the second and third arguments to **substring**. The rest of *thing* is ignored. *start* and *end* are ignored if *thing* is not a string.

If *junk-allowed* is non-nil, parsing stops without error if the syntax is invalid, and this function returns nil. The second value is then the index of the invalid character. If parsing is successful, the second value is the index of the place at which parsing was supposed to stop (*end*, or the end of *thing*). If *junk-allowed* is nil, invalid syntax signals an error.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host. If *thing* does not contain a manifest host name, then if *host* is non-nil, it is the host name to use, as a string. If *thing* is a string, a manifest host name may be at the beginning or the end, and consists of the name of a host followed by a colon. If *host* is nil then the host name is obtained from the default pathname in *defaults*. If *defaults* is not supplied, the default defaults (***default-pathname-defaults***) are used.

Note that if *host* is specified, and *thing* contains a host name, an error is signaled if they are not the same host.

fs:pathname-parse-error (fs:pathname-error error) *Condition*

This condition is signaled when **fs:parse-pathname** finds a syntax error in the string it is given.

fs:parse-pathname sets up a nonlocal proceed type **:new-pathname** for this condition. The proceed type expects one argument, a pathname, which is returned from **fs:parse-pathname**.

fs:merge-pathname-defaults *pathname* &optional *defaults default-type default-version*

Fills in unspecified components of *pathname* from the defaults and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, a symbol, or a Maclisp namelist. The returned value is always a pathname. The merging rules are documented on page 554.

If *defaults* is a pathname, rather than a defaults alist, then the defaults are taken from its components. This is how you merge two pathnames. (In Maclisp that operation is called **mergef**.)

defaults defaults to the value of ***default-pathname-defaults*** if unsupplied. *default-type* defaults to the value of **fs:*name-specified-default-type***. *default-version* defaults to **:newest**.

merge-pathnames *pathname* &optional *defaults* (*default-version* **:newest**)

Is the Common Lisp function for pathname defaulting. It does only some of the things that **fs:merge-pathname-defaults** does. It merges defaults from *defaults* (which defaults to the value of ***default-pathname-defaults***) into *pathname* to get a new pathname, which is returned. *pathname* can be a string (or symbol); then it is parsed and the result is defaulted. *default-version* is used as the version when pathname has a name but no version.

fs:merge-and-set-pathname-defaults *pathname* &optional *defaults default-type default-version*

This is the same as **fs:merge-pathname-defaults** except that after it is done the defaults-list *defaults* is modified so that the merged pathname is the new default. This is handy for programs that have sticky defaults, which means that the default for each command is the last filename used. (If *defaults* is a pathname rather than a defaults alist, then no storing back is done.) The optional arguments default the same way as in **fs:merge-pathname-defaults**.

These functions convert a pathname into a namestring for all or some of the pathname's components.

namestring *pathname*
Returns a string containing the printed form of *pathname*, as you would type it in. This uses the :string-for-printing operation.

file-namestring *pathname*
Returns a string showing just the name, type and version of *pathname*. This uses the :string-for-dired operation.

directory-namestring *pathname*
Returns a string showing just the device and directory of *pathname*. This uses the :string-for-directory operation.

enough-namestring *pathname* &optional *defaults*
Returns a string showing just the components of *pathname* which would not be obtained by defaulting from *defaults*. This is the shortest string that would suffice to specify pathname, given those defaults. It is made by using the :string-for-printing operation on a modified pathname.

This function yields a pathname given its components.

make-pathname &key (*defaults* t) *host device raw-device directory raw-directory name raw-name type raw-type version canonical-type original-type*
Returns a pathname whose components are as specified.

If *defaults* is a pathname or a defaults-alist, any components not explicitly specified default from it. If *defaults* is t (which is the default), then unspecified components default to nil, except for the host (since every pathname must have a specific host), which defaults based on *default-pathname-defaults*.

These functions give the components of a pathname.

pathname-host *pathname*
Returns the host component of *pathname*.

pathname-device *pathname*
pathname-directory *pathname*
pathname-name *pathname*
pathname-type *pathname*
pathname-version *pathname*
Likewise, for the other components.

These functions return useful information.

fs:user-homedir &optional *host reset-p* (*user* user-id) *force-p*
user-homedir-pathname &optional *host reset-p* (*user* user-id) *force-p*

Returns the pathname of the *user* 's home directory on *host*. These default to the logged in user and the host logged in to. Home directory is a somewhat system-dependent concept, but from the point of view of the Lisp Machine it is the directory where the user keeps personal files such as init files and mail.

This function returns a pathname without any name, type, or version component (those components are all nil).

If *reset-p* is specified non-nil, the machine the user is logged in to is changed to be *host*.

The synonym user-homedir-pathname is from Common Lisp.

init-file-pathname *program-name* &optional *host*

Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the login function.

These functions are useful for poking around.

fs:describe-pathname *pathname*

If *pathname* is a pathname object, this describes it, showing you its properties (if any) and information about files with that name that have been loaded into the machine. If *pathname* is a string, this describes all interned pathnames that match that string, ignoring components not specified in the string. One thing this is useful for is finding the directory of a file whose name you remember. Giving describe (see page 791) a pathname object invokes this function.

fs:pathname-plist *pathname*

Parses and defaults *pathname*, then returns the list of properties of that pathname.

fs:*pathname-hash-table* *Variable*

This is the hash table in which pathname objects are interned. You can find all pathnames ever constructed by applying the function maphash to this hash table.

24.5 Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the -*- line (see section 25.5, page 594) of a source file in the family. Several types of files with that name, in that directory, belong together. They are different members of the same family; for example, they may be source code and compiled code. However, there may be several other types of files that form a logically distinct group even though they have this same name; TEXT and PRESS for example. The exact mapping is done on a per host basis since it can sometimes be affected by host naming conventions.

The generic pathname of pathname *p* usually has the same host, device, directory, and name as *p* does. However, it has a version of :unspecific. The type of the generic pathname is obtained by sending a :generic-base-type *type-of-p* message to the host of *p*. The default response to this message is to return the associated type from fs:*generic-base-type-alist* if there is one, else *type-of-p*. Both the argument and the value are either strings, in interchange form, or canonical type symbols.

However, the ITS file system presents special problems. One cannot distinguish multiple generic base types in this same way since the type component does not exist as such; it is derived from the second filename, which unfortunately is also sometimes used as a version number. Thus, on ITS, the type of a generic pathname is always :unspecific if there is any association for the type of the pathname on fs:*generic-base-type-alist*.

Since generic pathnames are primarily useful for storing properties, it is important that they be as standardized and conceptualized as possible. For this reason, generic pathnames are defined to be backtranslated, i.e. the generic pathname of a pathname that is (or could be) the result of a logical host translation has the host and directory of the logical pathname. For example, the generic pathname of OZ:<L.WINDOW>;STREAM LISP would be SYS:WINDOW;STREAM U U if OZ is the system host.

All version numbers of a particular pathname share the same identical generic pathname. If the values of particular properties have changed between versions, it is possible for confusion to result. One way to deal with this problem is to have the property be a list associating version number with the actual desired property. Then it is relatively easy to determine which versions have which values for the property in question and select one appropriately. But in the applications for which generic pathnames are typically used, this is not necessary.

The :generic-pathname operation on a pathname returns its corresponding generic pathname. See page 563. The :source-pathname operation on a pathname returns the actual or probable pathname of the corresponding source file (with :newest as the version). See page 563.

fs:*generic-base-type-alist* *Variable*

This is an association list of the file types and the type of the generic pathname used for the group of which that file type is a part. Constructing a generic pathname replaces the file type with the association from this list, if there is one (except that ITS hosts always replace with :unspecific). File types not in this list are really part of the name in some sense. The initial list is

```
((:text . :text) ("DOC" . :text)
 (:press . :text) ("XGP" . :text)
 (:lisp . :unspecific) (:qfasl . :unspecific)
 (nil . :unspecific))
```

The association of :lisp and :unspecific is unfortunately made necessary by the problems of ITS mentioned previously. This way makes the generic pathnames of logically mapped LISP files identical no matter whether the logical host is mapped to an ITS host or not.

The first entry in the list with a particular cdr is the entry for the type that source files have. Note how the first element whose cdr is :unspecific is the one for :lisp. This is how the :source-pathname operation knows what to do, by default.

Some users may need to add to this list.

The system records certain properties on generic pathnames automatically.

:warnings This property is used to record compilation and other warnings for the file.

:definitions This property records all the functions and other things defined in the file. The value has one element for each package into which the file has been loaded; the element's car is the package itself and the cdr is a list of definitions made.

Each definition is a cons whose car is the symbol or function spec defined and whose cdr is the type of definition (usually one of the symbols defun, defvar, defflavor and defstruct).

:systems This property's value is a list of the names of all the systems (defined with defsystem, see page 660) of which this is a source file.

:file-id-package-alist

This property records what version of the file was most recently loaded. In case the file has been loaded into more than one package, as is sometimes necessary, the loaded version is remembered for each package separately. This is how make-system tells whether a file needs to be reloaded. The value is a list with an element for each package that the file has been loaded into; the elements look like

```
(package file-information)
```

package is the package object itself; *file-information* is the value returned by the :info operation on a file stream, and is usually a cons whose car is the truename (a pathname) and whose cdr is the file creation date (a universal time number).

Some additional properties are put on the generic pathname by reading the attribute list of the file (see page 597). It is not completely clear that this is the right place to store these properties, so it may change in the future. Any property name can appear in the attributes list and get onto the generic pathname; the standard ones are described in section 25.5, page 594.

24.6 Pathname Operations

This section documents the operations a user may send to a pathname object. Pathnames handle some additional operations that are only intended to be sent by the file system itself, and therefore are not documented here. Someone who wants to add a new host to the system would need to understand those internal operations.

The operations on pathnames that actually operate on files are documented in section 25.4, page 592. Certain pathname flavors, for specific kinds of hosts, allow additional special purpose operations. These are documented in section 24.7, page 568 in the section on the specific host type.

:generic-pathname *Operation on* pathname

Returns the generic pathname for the family of files of which this pathname is a member. See section 24.5, page 561 for documentation on generic pathnames.

:source-pathname *Operation on* pathname

Returns the pathname for the source file in the family of files to which this pathname belongs. The returned pathname has :newest as its version. If the file has been loaded in some fashion into the Lisp environment, then the pathname type is that which the user actually used. Otherwise, the conventional file type for source files is determined from the generic pathname.

:primary-device *Operation on* pathname

Returns the default device name for the pathname's host. This is used in generating the initial default pathname for a host.

Operations dealing with wildcards.

The character * in a namestring is a *wildcard*. It means that the pathname is a really a pattern which specifies a set of possible filenames rather than a single filename. The matches any sequence of characters within a single component of the name. Thus, the component FOO* would match FOO, FOOBAR, FOOT, or any other component starting with FOO.

Any component of a pathname can contain wildcards except the host; wild hosts are not allowed because a known host is required in order to know what flavor the pathname should be. If a pathname component is written in the namestring as just *, the actual component of the pathname instance is the keyword :wild. Components which contain wildcards but are not simply a single wildcard are represented in ways subject to change.

Pathnames whose components contain wildcards are called *wild* pathnames. Wild pathnames useful in functions such as delete-file for requesting the deletion of many files at once. Less obviously but more fundamentally, wild pathnames are required for most use of the function fs:directory-list; an entire directory's contents are obtained by specifying a pathname whose name, type and version components are :wild.

:wild-p *Operation on* pathname

Returns non-nil if this pathname contains any sort of wildcards. If the value is not nil, it is a keyword, one of device, :directory, :name, :type and :version, and it identifies the 'first' component which is wild.

:device-wild-p *Operation on* pathname

t if this pathname's device contains any sort of wildcards.

:directory-wild-p *Operation on* pathname
:name-wild-p *Operation on* pathname
:type-wild-p *Operation on* pathname
:version-wild-p *Operation on* pathname

Similar, for the other components that can be wild. (The host cannot ever be wild.)

:pathname-match *Operation on* pathname
candidate-pathname &optional (*match-host-p* **t**)

Returns t if candidate-pathname matches the pathname on which the operation is invoked (called, in this context, the *pattern pathname*). If the pattern pathname contains no wildcards, the pathnames match only if they are identical. This operation is intended in cases where wildcards are expected.

Wildcard matching is done individually by component; the operation returns t only if each component matches. Within each component, an occurrencs of * in pattern pathname's component can match any sequence of characters in *candidate-pathname*'s component. Other characters, except for host-specific wildcards, must match exactly. :wild as a component of the pattern pathname matches any component that *candidate-pathname* may have.

Note that if a component of the pattern pathname is nil, *candidate-pathname*'s component must be nil also to match it. Most user programs that read pathnames and use them as patterns default unspecified components to :wild first.

Examples:

```
(defvar pattern)
(defun test (str)
  (send pattern
        :pathname-match
        (parse-namestring str)))

(setq pattern
     (parse-namestring "OZ:*:<F*O>*.TEXT.*"))

(test "OZ:<FOO>A.TEXT") => t
(test "OZ:<FO>HAHA.TEXT.3") => t
(test "OZ:<FPPO>HAHA.TEXT.*") => t
(test "OZ:<FOX>LOSE.TEXT") => nil

(setq pattern
     (parse-namestring "OZ:*:<*>A.TEXT*.5"))

(test "OZ:<FOO>A.TEXT.5") => t
(test "OZ:<FOO>A.TEXTTTT.5") => t
(test "OZ:<FOO>A.TEXT") => nil
```

If *match-host-p* is nil, then the host components of the two pathnames are not tested. The result then depends only on the other components.

:translate-wild-pathname *Operation on* pathname
target-pattern starting-data &optional *reversible*

Returns a pathname corresponding to *starting-data* under the mapping defined by the wild pathnames *source-pattern*, which is the pathname this operation is invoked on, and *target-pattern*, the argument. It is expected that *starting-data* would match the source pattern under the :pathname-match operation.

:translate-wild-pathname is used by functions such as copy-file which use one wild pathname to specify a set of files and a second wild pathname to specify a corresponding filename for each file in the set. The first wild pathname would be used as the source-pattern and the second, specifying the name to copy each file to, would be passed as the *target-pattern* pathname.

Each component of the result is computed individually from the corresponding components of *starting-data* and the pattern pathnames, using the following rules:

1) If *target-pattern*'s component is ;wild, then the result component is taken from *starting-data*.

2) Otherwise, each non-wild character in *target-pattern*'s component is taken literally into the result. Each wild character in *target-pattern*'s component is paired with a wild character in *source-pattern*'s component, and thereby with the portion of *starting-data*'s component which that matched. This portion of *starting-data* appears in the result in place of the wild target character.

Example:

```
(setq source (fs:parse-pathname "OZ:PS:<FOO>A*B*.*.*"))
(setq target (fs:parse-pathname "OZ:SS:<*>*LOSE*.*B.*"))

(send source :translate-wild-pathname target
      (fs:parse-pathname "OZ:PS:<FOO>ALIBI.LISP.3"))
 => the pathname OZ:SS:<FOO>LILOSEI.LISPB.3
```

It is easiest to understand the mapping as being done in interchange case: the interchange components of the arguments are used and the results specify the interchange components of the value.

The type component is slightly special; if the *target-pattern* type is :wild, the canonical type of *starting-data* is taken and then interpreted according to the mappings of the target host. Example:

```
(setq source (fs:parse-pathname "OZ:PS:<FOO>A*.*.*"))
(setq target (fs:parse-pathname "U://usr//foo//b*.*"))

(send source :translate-wild-pathname target
      (fs:parse-pathname "OZ:PS:<FOO>ALL.LISP"))
 => the pathname U:/usr/foo/bll.l
```

If *reversible* is non-nil, rule 1 is not used; rule 2 controls all mapping. This mode is used by logical pathname translation. It makes a difference when the target pattern component is :wild and the source pattern component contains wildcards but is not simply :wild. For example, with source and target pattern components BIG* and *, and starting data BIGGER, the result is ordinarily BIGGER by rule 1, but with reversible translation the result is GER.

Operations to get a path name string out of a pathname object:

:string-for-printing — *Operation on* pathname

Returns a string that is the printed representation of the path name. This is the same as what you get if you princ the pathname or take string of it.

:string-for-wholine *length* — *Operation on* pathname

Returns a string like the :string-for-printing, but designed to fit in *length* characters. *length* is a suggestion; the actual returned string may be shorter or longer than that. However, the who-line updater truncates the value to that length if it is longer.

:string-for-editor — *Operation on* pathname

Returns a string that is the pathname with its components rearranged so that the name is first. The editor uses this form to name its buffers.

:string-for-dired *Operation on* pathname
Returns a string to be used by the directory editor. The string contains only the name, type, and version.

:string-for-directory *Operation on* pathname
Returns a string that contains only the device and directory of the pathname. It identifies one directory among all directories on the host.

:string-for-host *Operation on* pathname
Returns a string that is the pathname the way the host file system likes to see it.

Operations to move around through a hierarchy of directories:

:pathname-as-directory *Operation on* pathname
Assuming that the file described by the pathname is a directory, return another pathname specifying that *as* a directory. Thus, if sent to a pathname OZ:<RMS>FOO.DIRECTORY, it would return the pathname OZ:<RMS.FOO>. The name, type and version of the returned pathname are :unspecific.

:directory-pathname-as-file *Operation on* pathname
This is the inverse of the preceding operation. It returns a pathname specifying as a file the directory of the original pathname. The name, type and version of the original pathname are ignored.

The special symbol :root can be used as the directory component of a pathname on file systems that have a root directory.

Operations to manipulate the property list of a pathname:

:get *property-name* &optional *default-value* *Operation on* pathname
:getl *list-of-property-names* *Operation on* pathname
:putprop *value property-name* *Operation on* pathname
:remprop *property-name* *Operation on* pathname
:plist *Operation on* pathname
These manipulate the pathname's property list, and are used if you call the property list functions of the same names (see page 114) giving the pathname as the first argument. Please read the paragraph on page 546 explaining the care you must take in using property lists of pathnames.

24.7 Host File Systems Supported

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncracies. More host types may be added in the future.

24.7.1 ITS

An ITS pathname looks like "*host*: *device*: *dir*; *name type-or-version*". The primary device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: may be used.

ITS does not exactly fit the virtual file system model, in that a file name has two components (FN1 and FN2) rather than three (name, type, and version). Consequently to map any virtual pathname into an ITS filename, it is necessary to decide whether the FN2 is the type or the version. The rule is that usually the type goes in the FN2 and the version is ignored; however, certain types (LISP and TEXT) are ignored and instead the version goes in the FN2. Also if the type is :unspecific the FN2 is the version.

Given an ITS filename, it is converted into a pathname by making the FN2 the version if it is '<', '>', or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols :oldest and :newest, which correspond to '<' and '>' respectively.

In every ITS pathname either the version or the type is :unspecific or nil; sometimes both are. When you create a new ITS pathname, if you specify only the version or only the type, the one not specified becomes :unspecific. If both are specified, the version is :unspecific unless the type is a normally-ignored type (such as LISP) in which case the version is :newest and the type is :unspecific so that numeric FN2's are found.

Each component of an ITS pathname is mapped to upper case and truncated to six characters.

Special characters (space, colon, and semicolon) in a component of an ITS pathname can be quoted by prefixing them with right horseshoe (⊃) or equivalence sign (≡). Right horseshoe is the same character code in the Lisp Machine character set as control-Q in the ITS character set.

An ITS pathname can have a structured name, which is a list of two strings, the FN1 and the FN2. In this case there is neither a type nor a version.

An ITS pathname with an FN2 but no FN1 (i.e. a type and/or version but no name) is represented with the placeholder FN1 '⇔', because ITS pathname syntax provides no way to write an FN2 without an FN1 before it.

The ITS init file naming convention is "*homedir*; *user program*".

fs:*its-uninteresting-types* *Variable*

The ITS file system does not have separate file types and version numbers; both components are stored in the "FN2". This variable is a list of the file types that are "not important"; files with these types use the FN2 for a version number. Files with other types use the FN2 for the type and do not have a version number. The initial list is

```
        ("LISP" "TEXT" nil :unspecific)
```
Some users may need to add to this list.

:fn1 *Operation on* its-pathname
:fn2 *Operation on* its-pathname
These two operations return a string that is the FN1 or FN2 host-dependent component of the pathname.

:type-and-version *Operation on* pathname
:new-type-and-version *new-type new-version* *Operation on* pathname
These two operations provide a way of pretending that ITS pathnames can have both a type and a version. They use the first three characters of the FN2 to store a type and the last three to store a version number.

On an ITS-pathname, :type-and-version returns the type and version thus extracted (not the same as the type and version of the pathname). :new-type-and-version returns a new pathname constructed from the specified new type and new version.

On any other type of pathname, these operations simply return or set both the type component and the version component.

24.7.2 TOPS-20 (Twenex), Tenex, and VMS.

A pathname on TOPS-20 (better known as Twenex) looks like
host: *device*: <*directory*> *name*. *type*. *version*
The primary device is PS:.

TOPS-20 pathnames are mapped to upper case. Special characters (including lower-case letters) are quoted with the circle-cross (⊗) character, which has the same character code in the Lisp Machine character set as Control-V, the standard Twenex quoting character, in the ASCII character set.

If you specify a period after the name, but nothing after that, then the type is :unspecific, which translates into an empty extension on the TOPS-20 system. If you omit the period, you have allowed the type to be defaulted.

TOPS-20 pathnames allow the special version symbols :oldest and :newest. In the string form of a pathname, these are expressed as '.-2', and as an omitted version.

The directory component of a TOPS-20 pathname may be structured. The directory <FOO.BAR> is represented as the list ("FOO" "BAR").

The characters * and % are wildcards that match any sequence of characters and any single character (within one pathname component), respectively. To specify a filename that actually contains a * or % character, quote the character with ⊗. When a component is specified with just a single *, the symbol :wild appears in the pathname object.

The TOPS-20 init file naming convention is "<*user*>*program*.INIT".

When there is an attempt to display a TOPS-20 file name in the who-line and there isn't enough room to show the entire name, the name is truncated and followed by a center-dot character to indicate that there is more to the name than can be displayed.

Tenex pathnames are almost the same as TOPS-20 pathnames, except that the version is preceeded by a semi-colon instead of a period, the default device is DSK instead of PS, and the quoting requirements are slightly different.

VMS pathnames are basically like TOPS-20 pathnames, with a few complexities. The primary device is USRD$.

First of all, only alphanumeric characters are allowed in filenames (though $ and underscore can appear in device names).

Secondly, a version number is preceded by ';' rather than by '.'.

Thirdly, file types (called "extensions" in VMS terminology) are limited to three characters. Each of the system's canonical types has a special mapping for VMS pathnames, which is three characters long:

```
        :lisp → LSP      :text → TXT      :qfasl → QFS    :midas → MID
        :press → PRS     :widths → WID    :patch-directory → PDR
        :qwabl → QWB     :babyl → BAB     :mail → MAI     :xmail → XML
        :init → INI      :unfasl → UNF    :output → OUT
```

24.7.3 Unix and Multics Pathnames

A Unix pathname is a sequence of directory or file names separated by slashes. The last name is the filename; preceding ones are directory names (but directories are files anyway). There are no devices or versions. Alphabetic case is significant in Unix pathnames, no case conversion is normally done, and lower case is the default. Therefore, components of solid upper or lower case are inverted in case when going between interchange form and raw form. (What the user types in a pathname string is the raw form.)

Unix allows you to specify a pathname relative to your default directory by using just a filename, or starting with the first subdirectory name; you can specify it starting from the root directory by starting with a slash. In addition, you can start with '..' as a directory name one or more times, to refer upward in the hierarchy from the default directory.

Unix pathnames on the Lisp Machine provide all these features too, but the canonicalization to a simple descending list of directory names starting from the root is done on the Lisp Machine itself when you merge the specified pathname with the defaults.

If a pathname string starts with a slash, the pathname object that results from parsing it is called "absolute". Otherwise the pathname object is called "relative".

In an absolute pathname object, the directory component is either a symbol (nil, :unspecific or :root), a string, or a list of strings. A single string is used when there is only one level of directory in the pathname.

A relative pathname has a directory that is a list of the symbol :relative followed by some strings. When the pathname is merged with defaults, the strings in the list are appended to the strings in the default directory. The result of merging is always an absolute pathname.

In a relative pathname's string form, the string ".." can be used as a directory name. It is translated to the symbol :up when the string is parsed. That symbol is processed when the relative pathname is merged with the defaults.

Restrictions on the length of Unix pathnames require abbreviations for the standard Zetalisp pathname types, just as for VMS. On Unix the preferred mappings of all canonical types are one or two characters long. We give here the mappings in raw form; they are actually specified in interchange form.

```
:lisp → l          :text → tx       :qfasl → qf       :midas → md
:press → pr        :widths → wd     :patch-directory → pd
:qwabl → qw        :babyl → bb      :mail → ma        :xmail → xm
:init → in         :unfasl → uf     :output → ot
```

The Multics file system is much like the Unix one; there are absolute and relative pathnames, absolute ones start with a directory delimiter, and there are no devices or versions. Alphabetic case is significant.

There are differences in details. Directory names are terminated, and absolute pathnames begun, with the character '>'. The containing directory is referred to by the character '<', which is complete in itself. It does not require a delimiter. Thus, <<FOO>BAR refers to subdirectory FOO, file BAR in the superdirectory of the superdirectory of the default directory.

The limits on filename sizes are very large, so the system canonical types all use their standard mappings. Since the mappings are specified as upper case, and then interpreted as being in interchange form, the actual file names on Multics contain lower case.

24.7.4 Lisp Machine File Systems

There are two file systems that run in the MIT Lisp Machine system. They have different pathname syntax. Both can be accessed either remotely like any other file server, or locally.

The Local-File system uses host name LM for the machine you are on. A Local-File system on another machine can be accessed using the name of that machine as a host name, provided that machine is known as a file server.

The remainder of the pathname for the Local-File system looks like "*directory*; *name.type # version*". There is no restriction on the length of names; letters are converted to upper case. Subdirectories are allowed and are specified by putting periods between the directory components, as in RMS.SUBDIR;.

The TOPS-20 pathname syntax is also accepted. In addition, if the flag fs:*lmfs-use-twenex-syntax* is non-nil, Local-File pathnames print out using TOPS-20 syntax. Note that since the printed representation of a pathname is cached, changing this flag's value does not change the printing of pathnames with existing representations.

The Local-File system on the filecomputer at MIT has the host name FS.

The LMFILE system is primarily for use as a file server, unless you have 512k of memory. At MIT it runs on the filecomputer and is accessed remotely with host name FC.

The remainder of an LMFILE pathname looks like "*directory*; *name type # version*". However, the directory and name can be composed of any number of subnames, separated by backslashes. This is how subdirectories are specified. FOO;BAR\X refers to the same file as FOO\BAR;X, but the two ways of specifying the file have different consequences in defaulting, getting directory listings, etc.

Case is significant in LMFILE pathnames; however, when you open a file, the LMFILE system ignores the case when it matches your pathname against the existing files. As a result, the case you use matters when you create or rename a file, and appears in directory listings, but it is ignored when you refer to an existing file, and you cannot have two files whose names differ only in case. When components are accessed in interchange form, they are always converted to upper case.

24.7.5 Logical Pathnames

There is another kind of pathname that doesn't correspond to any particular file server. It is called a *logical* pathname, and its host is called a logical host. Every logical pathname can be translated into a corresponding *physical* pathname because each logical host records a corresponding actual ("physical") host and rules for translating the other components of the pathname.

The reason for having logical pathnames is to make it easy to keep bodies of software on more than one file system. An important example is the body of software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites may store the sources on an ITS file system, while others may store them on a TOPS-20. However, system software (including make-system) wishes to be able to find a particular file independent of the name of the host a particular site stores it on, or even the kind of host it is. This is done by means of the logical host SYS; all pathnames for system files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translations are different at each site. For example, at MIT the source files are stored on the TOPS-20 system named OZ, so MIT's site file says that SYS should translate to the host OZ.

Each logical host, such as SYS, has a list of translations, each of which says how to map certain pathnames for that host into pathnames for the corresponding physical host. To translate a logical pathname, the system tests each of the logical host's translations, in sequence, to see if it is applicable. (If none is applicable, an error is signaled.) A translation consists of a pair of pathnames or namestrings, typically containing wildcards. Unspecified components in them default to :wild. The *from*-pathname of the translation is used to match against the pathname to be translated; if it matches, the corresponding *to*-pathname is used to construct the translation, filling in its wild fields from the pathname being translated as in the :translate-wild-pathname operation (page 565).

Most commonly the translations contain pathnames that have only directories specified, everything else wild. Then the other components are unchanged by translation.

If the files accessed through the logical host are moved, the translations can be changed so that the same logical pathnames refer to the same files on their new physical host via physical pathnames changed to fit the restrictions and the conventions of the new physical host.

Each translation is specified as a list of two strings. The strings are parsed into pathnames and any unspecified components are defaulted to :wild. The first string of the pair is the source pattern; it is parsed with logical pathname syntax. The second string is the target pattern, and it is parsed with the pathname syntax for the specified physical host.

For example, suppose that logical host FOO maps to physical host BAR, a Tops-20, and has the following list of translations:

```
(("BACK;" "PS:<FOO.BACK>")
 ("FRONT;* QFASL" "SS:<FOO.QFASL>*.QFASL")
 ("FRONT;" "PS:<FOO.FRONT>"))
```

Then all pathnames with host FOO and directory BACK translate to host BAR, device PS and directory <FOO.BACK> with name, type and version unchanged. All pathnames with host FOO, directory FRONT and type QFASL translate to host BAR, device SS, directory <FOO.QFASL> and type QFASL, with name and version unchanged. All other pathnames with host FOO and directory FRONT map to host BAR, device PS and directory <FOO.FRONT>, with name, type and version unchanged. Note that the first translation whose pattern matches a given pathname is the one that is used.

Another site might define FOO's to map to a Unix host QUUX, with the following translation list:

```
(("BACK;" "//nd//foo//back//")
 ("FRONT;" "//nd//foo//front//"))
```

This site apparently does not see a need to store the QFASL files in a separate directory. Note that the slashes are duplicated to quote them for Lisp; the actual namestrings contain single slashes as is usual with Unix.

If the last translation's source pattern is entirely wild, it applies to any pathname not so far handled. Example:

```
(("BACK;" "//nd//foo//back//")
 ("" "//nd//fool//*//"))
```

Physical pathnames can also be *back-translated* into the corresponding logical pathname. This is the inverse transformation of ordinary translation. It is necessary to specify which logical host to back translate for, as it may be that the same physical pathname could be the translation of different logical pathnames on different hosts. Use the :back-translated-pathname operation, below.

fs:add-logical-pathname-host *logical-host physical-host translations*
fs:set-logical-pathname-host *logical-host* &key *physical-host translations*

Both create a new logical host named *logical-host*. Its corresponding physical host (that is, the host to which it should forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications, as described above. The two functions differ only in that one accepts positional arguments and the other accepts keyword arguments. Example:

```
(add-logical-pathname-host "MUSIC" "MUSIC-10-A"
      '(("MELODY;" "SS:<MELODY>")
        ("DOC;" "PS:<MUSIC-DOCUMENTATION>")))
```

This creates a new logical host called MUSIC. An attempt to open the file MUSIC:DOC;MANUAL TEXT 2 will be re-directed to the file MUSIC-10-A:PS:<MUSIC-DOCUMENTATION>MANUAL.TEXT.2 (assuming that the host MUSIC-10-A is a TOPS-20 system).

fs:make-logical-pathname-host *name*

Requests that the definition of logical host *name* be loaded from a standard place in the file system: namely, the file SYS: SITE; *name* TRANSLATIONS. This file is loaded immediately with load, in the fs package. It should contain code to create the logical host; normally, a call to fs:set-logical-pathname-host or fs:add-logical-pathname-host, above.

The same file is automatically reloaded, if it has been changed, at appropriate times: by load-patches, and whenever site information is updated.

:translated-pathname *Operation on* fs:logical-pathname

Converts a logical pathname to a physical pathname. It returns the translated pathname of this instance, a pathname whose host component is the physical host that corresponds to this instance's logical host.

If this operation is applied to a physical pathname, it simply returns that pathname unchanged.

:back-translated-pathname *pathname* *Operation on* fs:logical-pathname

Converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*. If *pathname* is not the translation of any logical pathname on this instance's host, nil is returned.

Here is an example of how this would be used in connection with truenames. Given a stream that was obtained by opening a logical pathname,

```
(send stream :pathname)
```

returns the logical pathname that was opened.

```
(send stream :truename)
```

returns the true name of the file that is open, which of course is a pathname on the physical host. To get this in the form of a logical pathname, one would do

```
(send (send stream :pathname)
      :back-translated-pathname
      (send stream :truename))
```

If this operation is applied to a physical pathname, it simply returns its argument. Thus the above example works no matter what kind of pathname was opened to create the stream.

fs:unknown-logical-pathname-translation (fs:pathname-error error) *Condition*

This is signaled when a logical pathname has no translation. The condition instance supports the :logical-pathname operation, which returns the pathname that was untranslatable.

The proceed type :define-directory is supported. It expects a single argument, a pathname or a string to be parsed into one. This defines the target pattern for a translation whose source pattern is the directory from the untranslatable pathname (and all else wild). Such a translation is added to the logical host, making it possible to translate the pathname.

A logical pathname looks like "*host*: *directory*; *name type version*". There is no way to specify a device; parsing a logical pathname always returns a pathname whose device component is :unspecific. This is because devices don't have any meaning in logical pathnames.

The equivalence-sign character (≡) can be used for quoting special characters such as spaces and semicolons. The double-arrow character ('↔') can be used as a place-holder for components that are nil, and the up-horseshoe ('∪') indicates :unspecific (generic pathnames typically have :unspecific as the type and the version). All letters are mapped to upper case unless quoted. The :newest, :oldest, and :wild values for versions are written as '>', '<', and '*' respectively.

There isn't any init file naming convention for logical hosts; you can't log into them. The :string-for-host, :string-for-wholine, :string-for-dired, and :string-for-editor messages are all passed on to the translated pathname, but the :string-for-printing is handled by the fs:logical-pathname flavor itself and shows the logical name.

24.7.6 Editor Buffer Pathnames

The hosts ED, ED-BUFFER and ED-FILE are used in pathnames which refer to buffers in the editor. If you open such a pathname, you get a stream that reads or writes the contents of an editor buffer. The three host names differ only in the syntax of the pathname, and in how it is interpreted.

The host ED is followed by an abbreviation that should complete to the name of an existing editor buffer. For example, the pathname ED:FOO could refer to the buffer FOO.LISP PS:<ME> OZ:.

The host ED-BUFFER is followed by an exact buffer name. If there is no buffer with that name, one is created. This is most useful for creating a buffer.

The host **ED-FILE** is followed by an arbitrary pathname, including a host name. An **ED-FILE** pathname refers to a buffer visiting that file. If necessary, the file is read into the editor. For example, **ED-FILE: OZ: PS:<ME>FOO.LISP** would refer to the same buffer as **ED: FOO**. The current default defaults are used in processing the pathname that follows **ED-FILE**, when the pathname is parsed.

24.8 Hosts

Each host known to the Lisp Machine is represented by a flavor instance known as a host object. The host object records such things as the name(s) of the host, its operating system type, and its network address(es). Host objects print like **#⊂FS:TOPS20-CHAOS-HOST "MIT-OZ"⊃**, so they can be read back in.

Not all hosts support file access. Those that do support it appear on the list **fs:*pathname-host-list*** and can be the host component of pathnames. A host object is also used as an argument when you make a Chaosnet connection for any purpose.

The hosts that you can use for making network connections appear in the value of **si:host-alist**. Most of the hosts you can use for pathnames are among these; but some, such as logical hosts, are not.

24.8.1 Parsing Hostnames

si:parse-host *namestring* &optional *no-error-p* (*unknown-ok* **t**)

Returns a host object that recognizes the specified name. If the name is not recognized, it is an error, unless *no-error-p* is non-**nil**; in that case, **nil** is returned.

If *unknown-ok* is non-**nil** (the default), a host table server on the local network is contacted, to see if perhaps it can find the name there. If it can't, an error is signalled or **nil** is returned, according to *no-error-p*. The host instance created in this manner contains all the kinds of information that a host defined from the host table file has.

If a string of the form **CHAOS|***nnn* is used, a host object is created and given *nnn* (interpreted as octal) as its Chaosnet address. This can be done regardless of the *unknown-ok* argument.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

sys:unknown-host-name *Condition*

(sys:local-network-error sys:network-error error)

This condition is signaled by **si:parse-host** when the host is not recognized, if that is an error.

The **:name** operation on the condition instance returns the string given to **si:parse-host**.

si:get-host-from-address *address network*

Returns a host object given an address and the name of the network which that address is for. Usually the symbol :chaos is used as the network name.

nil is returned if there is no known host with that address.

fs:get-pathname-host *name* &optional *no-error-p*

Returns a host object that can be used in pathnames. If the name is not recognized, it is an error, unless *no-error-p* is non-nil; in that case, nil is returned.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

si:parse-host and fs:get-pathname-host differ in the set of hosts searched.

fs:unknown-pathname-host (fs:pathname-error error) *Condition*

This condition is signaled by fs:get-pathname-host when the host is not recognized, if that is an error.

The :name operation on the condition instance returns the string given to fs:get-pathname-host.

fs:*pathname-host-list* *Variable*

This is a list of all the host objects that support file access.

si:host-alist *Variable*

This variable is a list of one element for each known network host. The element looks like this:

(*full-name host-object* (*nickname nickname2* ... *full-name*)
 system-type machine-type site
 network list-of-addresses network2 list-of-addresses2 ...)

The *full-name* is the host's official name. The :name operation on the host object returns this.

The *host-object* is a flavor instance that represents this host. It may be nil if none has been created yet; si:parse-host creates them when they are referred to.

The *nicknames* are alternate names that si:parse-host should recognize for this host, but which are not its official name.

The *system-type* is a symbol that tells what software the host runs. This is used to decide what flavor of host object to construct. Symbols now used include :lispm, :its, :tops-20, :tenex, :vms, :unix, :multics, :minits, :waits, :chaos-gateway, :dos, :rsx, :magicsix, :msdos, and others. Not all of these are specifically understood in any way by the Lisp Machine. If none of these applies to a host you wish to add, use a new symbol.

The *machine-type* is a symbol that describes the hardware of the host. Symbols in use include :lispm, :pdp10, :pdp11, :vax, :nu, :pe3230, and :ibmpc. (nil) has also been observed to appear here. Note that these machine types attempt to have wide meanings, lumping together various brands, models, etc.

The *site* does not describe anything about the host. Instead it serves to say what the Lisp Machine's site name was when the host was defined. This is so that, when a Lisp Machine system is moved to a different institution that has a disjoint set of hosts, all the old site's hosts can be deleted from the host alist by site reinitialization.

The *network*s and lists of addresses describe how to reach the host. Usually there is only one network and only one address in the list. The generality is so that hosts with multiple addresses on multiple networks can be recorded. Networks include :chaos and :arpa. The address is meaningful only to code for a specific network.

24.8.2 Host Object Operations

:name *Operation on host objects*
Returns the full, official name of the host.

:name-as-file-computer *Operation on host objects*
Returns the name to print in pathnames on this host (assuming it supports files). This is likely to be a short nickname of the host.

:short-name *Operation on host objects*
Returns the shortest known nickname for this host.

:pathname-host-namep *string* *Operation on host objects*
Returns t if *string* is recognized as a name for this host for purposes of pathname parsing. The local host will recognise LM as a pathname host name.

:system-type *Operation on host objects*
Returns the operating system type symbol for this host. See page 810.

:network-type *Operation on host objects*
Returns the symbol for one network that this host is connected to, or nil if it is not connected to any. :chaos is preferred if it is one of the possible values.

:network-typep *network* *Operation on host objects*
Returns t if the host is connected to the specified network.

:network-addresses *Operation on host objects*
Returns an alternating list of network names and lists of addresses, such as

```
(:chaos (3104) :arpa (106357002))
```

You can therefore find out all networks a host is known to be on, and its addresses on any network.

:sample-pathname *Operation on host objects*
Returns a pathname for this host, whose device, directory, name, type and version components are all nil. Sample pathnames are often useful because many file-system-dependent pathname operations depend only on the pathname's host.

`:open-streams` *Operation on host objects*
Returns a list of all the open file streams for files on this host.

`:close-all-files` *Operation on host objects*
Closes all file streams open for files on this host.

`:generic-base-type` *type-component* *Operation on host objects*
Returns the type component for a generic pathname assuming it is being made from a pathname whose type component is the one specified.

25. Accessing Files

The Lisp Machine can access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Lisp Machine itself, if the machine has its own file system. You are not allowed to refer to files without first logging in, and you may also need to specify a username and password for the host on which the file is stored; see page 801.

The way to read or write a file's contents is to *open* the file to get an input or output stream, use the standard stream I/O functions or operations described in chapters 22 and 23, and then close the stream. The first section of this chapter tells how to open and close the stream. The rest of the chapter describes things specific to files such as deleting and renaming, finding out the true name of the file that has been opened, and listing a directory.

Files are named with *pathnames*. There is much to know about pathnames aside from accessing files with them; all this is described in the previous chapter.

Many functions in this chapter take an argument called *file* which is intended to specify a file to be operated on. This argument may be given as a pathname (which is defaulted), a namestring (which is parsed into a pathname and then defaulted), or a stream open to a file (the same file is used).

25.1 Opening and Closing File Streams

with-open-file (*stream file options...*) *body...* *Macro*

Evaluates the *body* forms with the variable *stream* bound to a stream that reads or writes the file named by the value of *file*. The *options* forms evaluate to the file-opening options to be used; see page 582.

When control leaves the body, either normally or abnormally (via throw), the file is closed. If a new output file is being written and control leaves abnormally, the file is aborted and it is as if it were never written. Because it always closes the file, even when an error exit is taken, with-open-file is preferred over open. Opening a large number of files and forgetting to close them tends to break some remote file servers, ITS's for example.

If an error occurs in opening the file, the result depends on the values of the *error* option, the *if-exists* option, and the *if-does-not-exist* option. An error may be signaled (and possibly corrected with a new pathname), or *stream* may be bound to a condition object or even nil.

with-open-file-case (*stream file options...*) *clauses...* *Macro*

This opens and closes the file like with-open-file, but what happens afterward is determined by *clauses* that are like the clauses of a condition-case (page 702). Each clause begins with a condition name or a list of condition names and is executed if open signals a condition that possesses any of those names. A clause beginning with the symbol :no-error is executed if the file is opened successfully. This would be where the reading

or writing of the file would be done.
Example:
```
    (with-open-file-case (stream (send generic-pathname
                                        :source-pathname))
      (sys:remote-network-error (format t "~&Host down."))
      (fs:file-not-found (format t "~&(New file)"))
      (:no-error (setq list (read stream))))
```

file-retry-new-pathname (*pathname-var condition-names...*) *body...* *Macro*
file-retry-new-pathname-if *Macro*
cond-form (*pathname-var condition-names...*) *body...*

file-retry-new-pathname executes *body*. If *body* does not signal any of the conditions in *condition-names*, *body*'s values are simply returned. If any of *condition-names* is signaled, file-retry-new-pathname reads a new pathname, setq's *pathname-var* to it, and executes *body* again.

The user can type End instead of a pathname if he wishes to let the condition be handled by the debugger.

file-retry-new-pathname-if is similar, but the conditions are handled only if *cond-form*'s value is non-nil.

For an example, see the example of the following macro.

with-open-file-retry *Macro*
(*stream* (*pathname-var condition-names...*) *options...*) *body...*

Like with-open-file inside of a file-retry-new-pathname. If an error occurs while opening the file and it has one of the specified *condition-names*, a new pathname is read, the variable *pathname-var* is setq'd to it, and another attempt is made to open a file with the newly specified name. Example:
```
    (with-open-file-retry (instream (infile fs:file-not-found))
      ...)
```
infile should be a variable whose value is a pathname or namestring. The example is equivalent to
```
    (file-retry-new-pathname (infile fs:file-not-found)
      (with-open-file (instream infile)
        ...))
```

with-open-file-search *Macro*

Opens a file, trying various pathnames until one of them succeeds. The pathnames tried differ only in their type components. For example, load uses this macro to search for either a compiled file or a source file. The calling sequence looks like
```
    (with-open-file-search
        (streamvar (operation defaults auto-retry)
                   types-and-pathname options...)
      body...)
```

with-open-file-search tries opening various files until one succeeds; then binds *streamvar* to the stream and executes *body*, closing the stream on exit. The values of *body* are returned.

types-and-pathname specifies which files to open. It should be a form which evaluates to two values, the first being a list of types to try and the second being a pathname called the base pathname. Each pathname to try is made by merging the base pathname with the defaults *defaults* and one of the types. The types may be strings or canonical type keywords (see section 24.2.3, page 551).

options are forms whose values should be alternating to keywords and values, which are passed to **open** each time.

If all the names to be tried fail, a **fs:multiple-file-not-found** error is signaled. *operation* is provided just so that the **:operation** operation on the condition object can return it. Usually the value given for *operation* should be the user-level function for which the **with-open-file-search** is being done.

If *auto-retry* is non-nil, an error causes the user to be prompted for a new base pathname. The entire set of types specified is tried anew with the new pathname.

open *file* &rest *options*
Returns a stream that is connected to the specified file. Unlike Maclisp, the **open** function creates streams only for *files*; streams of other kinds are created by other functions. The *file* and *options* arguments are the same as in **with-open-file**; see above.

When the caller is finished with the stream, it should close the file by using the **:close** operation or the **close** function. The **with-open-file** special form does this automatically and so is usually preferred. **open** should only be used when the control structure of the program necessitates opening and closing of a file in some way more complex than the simple way provided by **with-open-file**. Any program that uses **open** should set up **unwind-protect** handlers (see page 82) to close its files in the event of an abnormal exit.

close *stream* &optional *option*
The **close** function simply sends the **:close** message to *stream*. If *option* is **:abort** for a file output stream, the file is discarded.

cli:close *stream* &key *abort*
The Common Lisp version of **close** is the same as **close** except for its calling convention. If *abort* is non-nil for a file output stream, the file is discarded.

fs:close-all-files
Closes all open files. This is useful when a program has run wild opening files and not closing them. It closes all the files in **:abort** mode (see page 464), which means that files open for output are deleted. Using this function is dangerous, because you may close files out from under various programs like Zmacs and ZMail; only use it if you have to and if you feel that you know what you're doing.

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. In addition, for compatibility with the Maclisp **open** function, if only a single option is specified it is either a keyword or a list of keywords (not alternating with values).

The file-opening options control things like whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, etc.

The following keyword arguments are standardly recognized; additional keywords can be implemented by particular file system hosts.

direction Controls which direction of I/O can be done on the resulting stream. The possible values are **:input** (the default), **:output**, **nil**, **:probe**, **:probe-directory** and **:probe-link**. The first two should be self-explanatory. **nil** or **:probe** means that this is a "probe" opening; no data are to be transferred, the file is being opened only to verify its existence or access its properties. The stream created in this case does not permit any I/O. **nil** and **:probe** differ in causing different defaults for the argument *if-does-not-exist*. If that argument is specified explicitly, **nil** and **:probe** are equivalent.

:probe-directory is used to see whether a directory exists. If the directory specified for the file to be opened is found, then the **open** completes (returning a non-I/O stream) as if the specified file existed whether it really exists or not.

:probe-link is used to find out the truename of a link. If the file specified exists as a link, then the **open** completes returning a non-I/O stream which describes the link itself rather than the file linked to. If the file exists and is not a link, the **open** also completes for it as with any probe.

Common Lisp defines the value **:io** for this argument, requesting a stream that can do input and output, but no file system supported by the Lisp Machine has this capability.

characters The possible values are **t** (the default), **nil**, which means that the file is a binary file, and **:default**, which means that the file system should decide whether the file contains characters or binary data and open it in the appropriate mode.

byte-size The possible values are **nil** (the default), a number, which is the number of bits per byte, and **:default**, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, **nil** selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, **nil** selects the default byte size of 16 bits.

element-type This is the Common Lisp way to specify what kind of objects the stream wants to read or write. This combines the effect of the *characters* and *byte-size* arguments. The value is a type specifier; it must be one of the following:

string-char Read or write characters as usual. The default.

character Read or write characters, dealing with characters that are more than 8 bits. You can succeed in writing out any sequence of character objects and reading it back, but the file does not look anything like a text file.

(unsigned-byte *n*)
: Read or write *n*-bit bytes. Like *characters* = nil, *byte-size* = *n*.

unsigned-byte
: Similar, but uses the byte size that the file was originally written with. This is the same as *characters* = nil, *byte-size* = :default.

(signed-byte *n*)
: Read or write *n*-bit bytes, sign-extending on input. Each byte read from the file is sign-extended so that its most significant bit serves as a sign bit.

signed-byte
: Similar, but uses the byte size that the file was originally written with.

(mod *n*)
: Like unsigned-byte for a big enough byte size to hold all numbers less than *n*. bit is also accepted, and means (mod 2).

:default
: Is allowed, even though it is not a type specifier. It is the same as using :default as the value of *characters*.

if-exists
: For output opens, *if-exists* specifies what to do if a file with the specified name already exists. There are several values you can use:

:new-version
: Create a new version. This makes sense only when the pathname has :newest as its version, and it is the default in that case.

:supersede
: Make a new file which, when closed, replaces the old one.

:overwrite
: Write over the data of the existing file, starting at the beginning, and set the file's length to the length of the newly written data.

:truncate
: Like :overwrite except that it discards the old contents of the file immediately, making it empty except for what is written into it this time.

:append
: Add new data onto the existing file at the end.

:rename
: Rename the existing file and then create a new one.

:rename-and-delete
: Rename the existing file, create a new one, and delete the old file when the new one is closed.

:error
: Signal an error (fs:file-already-exists). This is the default when the pathname's version is not :newest. The further handling of the error is controlled by the *error* argument.

nil
: Return nil from open in this case. The *error* argument is irrelevant in this case.

if-does-not-exist
: Specifies what to do when the file requested does not exist. There are three allowed values:

:create	Create a file. This is the default for output opens, except when *if-exists* is :append, :overwrite or :truncate. This silly exception is part of the Common Lisp specifications.
:error	Signal an error. This is the default for input opens, and also for output opens when *if-exists* is :append, :overwrite or :truncate. The further handling of the error is controlled by the *error* argument.
nil	Return nil from open. This is the default for :probe opens. The *error* argument is irrelevant in this case.

error Specifies what to do if an error is signaled for any reason. (Note that the values of the *if-exists* and *if-does-not-exist* arguments control whether an error is signaled in certain circumstances.) The possible values are t (the default), :reprompt and nil. t means that nothing special is done, so the error invokes the debugger if the caller does not handle it. nil means that the condition object should be returned as the value of open. :reprompt means that a new file name should be read and opened.

Any caller which need not know reliably which file was ultimately opened might as well specify :reprompt for this argument. Callers which need to know if a different file is substituted should never specify :reprompt; they may use with-open-file-retry or file-retry-new-pathname (see page 581) if they wish to permit an alternative file name to be substituted.

:submit If specified as t when opening a file for output, the file is submitted as a batch job if it is closed normally. The default is nil. You must specify :direction :output as well.

deleted The default is nil. If t is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.

temporary If t is specified, the file is marked as temporary, if the file system has that concept. The default is nil.

preserve-dates If t is specified, the file's reference and modification dates are not updated. The default is nil.

flavor This controls the kind of file to be opened. The default is nil, a normal file. Other possible values are :directory and :link. Only certain file systems recognize this keyword.

link-to When creating a file with *flavor* :link, this argument must be specified; its value is a pathname or namestring that becomes the target of the link.

submit The value can be either nil (the default) or t. If the value is t, and the :direction is :output, the resulting file will be submitted as a batch job. Currently, this option is implemented only for Twenex and VMS.

estimated-size The value may be nil (the default), which means there is no estimated size, or a number of bytes. Some file systems use this to optimize disk allocation.

physical-volume The value may be nil (the default), or a string that is the name of a physical volume on which the file is to be stored. This is not meaningful for all file systems.

logical-volume The value may be nil (the default), or a string that is the name of a logical volume on which the file is to be stored. This is not meaningful for all file systems.

super-image The value may be nil (the default), or t, which disables the special treatment of rubout in ASCII files. Normally, rubout is an escape which causes the following character to be interpreted specially, allowing all characters from 0 through 376 (octal) to be stored. This applies to ASCII file servers only.

raw The value may be nil (the default), or t, which disables all character set translation in ASCII files. This applies to ASCII file servers only.

In the Maclisp compatibility mode, there is only one *option*, and it is either a symbol or a list of symbols. These symbols are recognized no matter what package they are in, since Maclisp does not have packages. The following symbols are recognized:

in, read Select opening for input (the default).

out, write, print
Select opening for output; a new file is to be created.

binary, fixnum Select binary mode; otherwise character mode is used. Note that fixnum mode uses 16-bit binary words and is not compatible with Maclisp fixnum mode, which uses 36-bit words. On the PDP-10, fixnum files are stored with two 16-bit words per PDP-10 word, left-justified and in PDP-10 byte order.

character, ascii
The opposite of fixnum. This is the default.

single, block Ignored for compatibility with the Maclisp open function.

byte-size Must be followed by a number in the options list, and must be used in combination with fixnum. The number is the number of bits per byte, which can be from 1 to 16. On a PDP-10 file server these bytes will be packed into words in the standard way defined by the ILDB instruction. The :tyi stream operation will (of course) return the bytes one at a time.

probe, error, noerror, raw, super-image, deleted, temporary
These are not available in Maclisp. The corresponding keywords in the normal form of file-opening options are preferred over these.

25.2 File Stream Operations

The following functions and operations may be used on file streams, in addition to the normal I/O operations which work on all streams. Note that several of these operations are useful with file streams that have been closed. Some operations use pathnames; refer to chapter 24, page 545 for an explanation of pathnames.

file-length *file-stream*

Returns the length of the file open on *file-stream*, in terms of the units in which I/O is being done on that stream. (A stream is needed, rather than just a pathname, in order to specify the units.)

file-position *file-stream* &optional *new-position*

With one argument, returns the current position in the file of *file-stream*, using the :read-pointer stream operation. It may return nil meaning that the position cannot be determined. In fact, it always returns nil for a stream open in character mode and not at the beginning of the file.

With two arguments, sets the position using the :set-pointer stream operation, if possible, and returns t if the setting was possible and nil if not. You can specify :start as the *new-position* to position to the beginning of the file, or :end to position to the end.

:pathname *Operation on file streams*

Returns the pathname that was opened to get this stream. This may not be identical to the argument to open, since missing components will have been filled in from defaults. The pathname may have been replaced wholesale if an error occurred in the attempt to open the original pathname.

:truename *Operation on file streams*

Returns the pathname of the file actually open on this stream. This can be different from what :pathname returns because of file links, logical devices, mapping of version :newest to a particular version number, etc. For an output stream the truename is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:generic-pathname *Operation on file streams*

Returns the generic pathname of the pathname that was opened to get this stream. Normally this is the same as the result of sending the :generic-pathname message to the value of the :pathname operation on the stream; however, it does special things when the Lisp system is bootstrapping itself.

:qfaslp *Operation on file streams*

Returns t if the file has a magic flag at the front that says it is a QFASL file, nil if it is an ordinary file.

:length *Operation on file streams*

Returns the length of the file, in bytes or characters. For text files on ASCII file servers, this is the number of ASCII characters, not Lisp Machine characters. The numbers are different because of character-set translation; see section 25.8, page 607 for a full explanation. For an output stream the length is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:creation-date *Operation on file streams*
Returns the creation date of the file, as a number that is a universal time. See the chapter on the time package (chapter 34, page 776).

:info *Operation on file streams*
Returns a cons of the file's truename and its creation date. This can be used to tell if the file has been modified between two **open**'s. For an output stream the information is not guaranteed to be correct until after the stream has been closed.

:properties &optional (*error-p* **t**) *Operation on file streams*
This returns two values: a property list (like an element of the list returned by **fs:directory-list**), and a list of the settable properties. See the section on standard file properties (section 25.6, page 598) for a description of the ones that may possible found in the list.

:set-byte-size *new-byte-size* *Operation on file streams*
This is only allowed on binary file streams. The byte size can be changed to any number of bits from 1 to 16.

:delete &optional (*error-p* **t**) *Operation on file streams*
Deletes the file open on this stream. For the meaning of *error-p*, see the **deletef** function. The file doesn't really go away until the stream is closed.

:undelete &optional (*error-p* **t**) *Operation on file streams*
If you have used the **:deleted** option in **open** to open a deleted file, this operation undeletes the file.

:rename *new-name* &optional (*error-p* **t**) *Operation on file streams*
Renames the file open on this stream. For the meaning of *error-p*, see the **renamef** function.

File output streams implement the **:finish** and **:force-output** operations.

25.3 Manipulating Files

This section describes functions for doing things to files aside from reading or writing their contents.

truename *object*
Returns the truename of the file specified somehow by *object*. If *object* is a plausible stream, it is asked for the truename with the **:truename** operation. Otherwise, *object* is converted to a pathname and that pathname is opened to get its file's truename.

delete-file *file* &key (*error-p* t) *query?*
deletef *file* &optional (*error-p* t) *query?*

Both delete the specified file. The two functions differ in accepting keyword arguments versus positional arguments. *file* may contain wildcard characters, in which case multiple files are deleted.

If *query?* is non-nil, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually deleted.

If *error-p* is t, then if an error occurs it is signaled as a Lisp error. If *error-p* is nil and an error occurs, the error message is returned as a condition object. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the truename of the file, and the cadr is non-nil if the file was actually deleted (it is always t unless querying was done).

undelete-file *file* &key (*error-p* t) *query?*
undeletef *file* &optional (*error-p* t) *query?*

Both undelete the specified file. Wildcards are allowed, just as in deletef. The rest of the calling conventions are the same as well. The two functions differ in taking keyword arguments versus positional arguments.

Not all file systems support undeletion, and if it is not supported on the one you are using, it gets an error or returns a string according to *error-p*. To find out whether a particular file system supports this, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion.

rename-file *file new-name* &key (*error-p* t) *query?*
renamef *file new-name* &optional (*error-p* t) *query?*

Both rename the specified file to *new-name* (a pathname or string). The two functions differ in taking keyword arguments versus positional arguments. *file* may contain wildcards, in which case multiple files are renamed. Each file's new name is produced by passing *new-name* to merge-pathname-defaults with the file's truename as the defaults. Therefore, *new-name* should be a string in this case.

If *query?* is non-nil, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually renamed.

If *error-p* is t, then if an error occurs it is signaled as a Lisp error. If *error-p* is nil and an error occurs, the error message is returned as a condition object. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the original truename of the file, the cadr is the name it was to be renamed to, and the caddr is non-nil if the file was renamed. The caddr is nil if the user was queried and said no.

copy-file *file new-name* &key (*error* t) (*copy-creation-date* t) (*copy-author* t) *report-stream* (*create-directories* :query) (*characters* :default) (*byte-size* :default)

Copies the file specified by *file* to the name *new-name*.

characters and *byte-size* specify what mode of I/O to use to transfer the data. *characters* can be

t	to specify character input and output.
nil	for binary input and output,
:ask	meaning ask the user which one
:maybe-ask	meaning ask if it is not possible to tell with certainty which method is best,
:default	meaning to guess as well as possible automatically.

If binary transfer is done, *byte-size* specifies the byte size to use; :default means to ask the file system for the byte size that the old file is stored in, just as it does in **open**.

copy-author and *copy-creation-date* say whether to set those properties of the new file to be the same as those of the old file. If a property is not copied, it is set to your login name or the current date and time.

report-stream, if non-nil, is a stream on which a message should be printed describing the file copied, where it is copied to, and which mode was used.

create-directories says what to do if the output filename specifies a directory that does not exist. It can be t meaning create the directory, nil meaning treat it as an error, or :query meaning ask the user which one to do. The default is :query.

error, if nil, means that if an error happens then this function should just return an error indication.

If the pathname to copy from contains wildcards, multiple files are copied. The new name for each file is obtained by merging *new-name* (parsed into a pathname) with that file's truename as a default. The mode of copy is determined for each file individually, and each copy is reported on the *report-stream* if there is one. If *error* is nil, an error in copying one file does not prevent the others from being copied.

There are four values. If wildcards were used, each value is a list with one element describing each file that matched; otherwise, each value describes the single file specified (though the value may be a list anyway). The values, for each file, are:

output-file	The defaulted pathname to be opened for output in copying this file.
truename	The truename of the file copied
outcome	The truename of the new file, If the file was successfully copied. A condition object, if there was an error and *error* was nil. nil if the user was asked whether to copy this file and said no.
mode	A Common Lisp type descriptor such as string-char or (unsigned-byte 8) saying how the file was copied.

probe-file *file*
probef *file*

Returns **nil** if there is no file named *file*; otherwise returns a pathname that is the true name of the file, which can be different from *file* because of file links, version numbers, etc. If *file* is a stream, this function cannot return **nil**.

Any problem in opening the file except for **fs:file-not-found** signals an error.

probef is the Maclisp name; **probe-file** is the Common Lisp name.

file-write-date *file*

Returns the creation date/time of *file*, as a universal time.

file-author *file*

Returns the name of the author of *file* (the user who wrote it), as a string.

viewf *file* &optional (*output-stream* ***standard-output***) *leader*

Copies the contents of the specified file, opened in character mode, onto **output-stream**. Normally this has the effect of printing the file on the terminal. *leader* is passed along to **stream-copy-until-eof** (see page 457).

fs:create-link *link-name link-to* &key (*error* **t**)

Creates a link named *link-name* which points to a file named *link-to*. An error happens if the host specified in *link-name* does not support links, or for any of the usual problems that can happen in creating a file.

25.3.1 Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as **defun** and **defvar** which define the functions and variables of the program.

Loading a compiled (or QFASL) file is similar, except that the file does not contain text but rather pre-digested expressions created by the compiler which can be loaded more quickly.

These functions are for loading single files. There is a system for keeping track of programs which consist of more than one file; for further information refer to chapter 28, page 660.

load *file* &key *verbose print* (*if-does-not-exist* **t**) *set-default-pathname package*

Loads the specified file into the Lisp environment. If *file* is a stream, **load** reads from it; otherwise *file* is defaulted from the default pathname defaults and the result specifies a file to be opened. If the file is a QFASL file, **fasload** is used; otherwise **readfile** is used. If *file* specifies a name but no type, **load** looks first for the canonical type **:qfasl** and then for the canonical type **:lisp**.

Normally the file is read into the package specified in its attribute list, but if *package* is supplied then the file is read in that package. If *package* is **nil** and *verbose* is **nil**, **load** prints a message saying what file is being loaded and what package is being used. *verbose* defaults to the value of ***load-verbose***.

If *if-does-not-exist* is nil, load just returns nil if no file with the specified name exists. Error conditions other than fs:file-not-found are not handled by this option.

If a file is loaded, **load** returns the file's truename.

If *print* is non-nil, the value of each expression evaluated from the file is printed on *standard-output*.

pathname is defaulted from the default pathname defaults. If *set-default-pathname* is non-nil, the pathname defaults are set to the name of the file loaded. The default for *set-default-pathname* is t.

load used to be called with a different calling sequence:

`(load` *pathname pkg nonexistent-ok dont-set-default*`)`

This calling sequence is detected and still works, but it is obsolete.

`*load-verbose*` *Variable*

Is the default value for the *verbose* argument to **load**.

`readfile` *file* &optional *pkg no-msg-p*

readfile is the version of **load** for text files. It reads and evaluates each expression in the file. As with **load**, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package.

`fasload` *file* &optional *pkg no-msg-p*

fasload is the version of **load** for QFASL files. It defines functions and performs other actions as directed by the specifications inserted in the file by the compiler. As with **load**, *pkg* can specify what package to read the file into. Unless *no-msg-p* is **t**, a message is printed indicating what file is being read into what package.

25.4 Pathname Operations That Access Files

Here are the operations that access files. Many accept an argument *error* or *error-p* which specifies whether to signal an error or to return a condition instance, if the file cannot be accessed. For these arguments, nil and non-nil are the only significant values. :reprompt has no special meaning as a value. That value when passed to one of the file accessing functions (**open**, deletef, etc.) has its special significance at a higher level.

`:truename` *Operation on* pathname

Returns a pathname object describing the exact name of the file specified by the pathname the object is sent to.

This may be different from the original pathname. For example, the original pathname may have :newest as the version, but the truename always has a number as the version if the file system supports versions.

:open *pathname* &rest *options* *Operation on* pathname

Opens a stream for the file named by the pathname. The argument *pathname* is what the :pathname operation on the resulting stream should return. When a logical pathname is opened, *pathname* is that logical pathname, but **self** is its translated pathname.

options is a list of alternating keywords and values, as would be passed to **open**. The old style of **open** keywords are not allowed; when they are used with **open**, **open** converts them to the new style before sending the :open message.

:delete &optional (*error-p* **t**) *Operation on* pathname
:undelete &optional (*error-p* **t**) *Operation on* pathname

Respectively delete or undelete the file specified by the pathname.

All file systems support :delete but not all support :undelete.

If *error-p* is **nil**, problems such as nonexistent files cause a string describing the problem to be returned. Otherwise, they signal an error.

:undeletable-p *Operation on* pathname

Returns **t** if this pathname is for a file system which allows deletion to be undone. Such pathnames support the :undelete and :expunge operations.

:rename *new-name* &optional (*error-p* **t**) *Operation on* pathname

Renames the file specified by the pathname. *new-name*, a string or pathname, specifies the name to rename to. If it is a string, it is parsed using **self** as the defaults.

If *error-p* is **nil**, problems such as nonexistent files cause a string describing the problem to be returned. Otherwise, they signal an error.

:complete-string *string options* *Operation on* pathname

Attempts to complete the filename *string*, returning the results. This operation is used by the function **fs:complete-pathname** (see page 602). The pathname the message is sent to is used for defaults. *options* is a list whose elements may include :deleted, :read (file is for input), :write (it's for output), :old (only existing files allowed), or :new-ok (new files are allowed too).

There are two values: a string, which is the completion as far as possible, and a flag, which can be :old, :new or **nil**. :old says that the returned string names an existing file, :new says that the returned string is no file but some completion was done, **nil** says that no completion was possible.

:change-properties *error-p* &rest *properties* *Operation on* pathname

Changes the properties of the file specified by the pathname. *properties* should be an alternating list of property names and values.

:directory-list *options* *Operation on* pathname
Performs the work of (fs:directory-list *this-pathname options*...).

:properties *Operation on* pathname
Returns a property list (in the form of a directory-list element) and a list of settable properties. See section 25.6, page 598 for more information on file properties.

:wildcard-map *function plistp dir-list-options* &rest *args* *Operation on* pathname
Maps *function* over all the files specified by this pathname (which may contain wildcards). Each time *function* is called, its first argument is a pathname with no wildcards, or else a directory-list element (whose car is a pathname and whose cdr contains property names and values). The elements of *args* are given to *function* as additional arguments.

plistp says whether *function*'s first argument should be a directory-list element or just a pathname. t specifies a directory-list element. That provides more information, but it makes it necessary to do extra work if the specified pathname does *not* contain wildcards.

dir-list-options is passed to fs:directory-list. You can use this to get deleted files mentioned in the list, for example.

The remaining file-access operations are defined only on certain file systems.

:expunge &key (*error* t) *Operation on* pathname
Expunges the directory specified by the host, device and directory components of the pathname.

The argument *error* says whether to signal an error if the directory does not exist. nil means just return a string instead.

:create-directory &key (*error* t) *Operation on* pathname
Creates the directory specified in this pathname.

:remote-connect &key (*error* t) *access* *Operation on* pathname
Performs the work of fs:remote-connect with the same arguments on this pathname's host.

25.5 File Attribute Lists

Any text file can contain an *attribute list* that specifies several attributes of the file. The above loading functions, the compiler, and the editor look at this property list. Attribute lists are especially useful in program source files, i.e. a file that is intended to be loaded (or compiled and then loaded). QFASL files also contain attribute lists, copied from their source files.

If the first non-blank line in a text file contains the three characters '-*-', some text, and '-*-' again, the text is recognized as the file's attribute list. Each attribute consists of the attribute name, a colon, and the attribute value. If there is more than one attribute they are separated by semicolons. An example of such an attribute list is:

```
    ; -*- Mode:Lisp; Package:Cellophane; Base:10 -*-
```

This defines three attributes: mode, package, and base. The initial semicolon makes the line look like a comment rather than a Lisp expression. Another example is:

```
    .c Part of the Lisp Machine manual.  -*- Mode:Bolio -*-
```

An attribute name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. An attribute value can be such a name, or a decimal number, or several such items separated by commas. Spaces may be used freely to separate tokens. Upper and lower-case letters are not distinguished. There is *no* quoting convention for special characters such as colons and semicolons.

If the attribute list text contains no colons, it is an old Emacs format, containing only the value of the Mode attribute.

The file attribute list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret. The Emacs editor on the PDP-10 knows how to interpret these attribute lists (primarily in order to look at the Mode attribute).

The Lisp Machine handles the attribute list stored in the file by parsing it into a Lisp data structure, a property list. Attribute names are interpreted as Lisp symbols and are interned on the keyword package. Numbers are interpreted as Lisp fixnums and are read in decimal. If a attribute value contains any commas, then the commas separate several expressions that are formed into a list.

When a file is compiled, its attribute list data structure is stored in the QFASL file. It can be loaded back from the QFASL file as well. The representation in the QFASL file resembles nothing described here, but when the attribute list is extracted from there, the same Lisp data structure described above is obtained.

When a file is edited, loaded, or compiled, its file attribute list is read in and the properties are stored on the property list of the generic pathname (see section 24.5, page 561) for that file, where they can be retrieved with the :get and :plist messages. This is done using the function fs:read-attribute-list, below. So the way you examine the properties of a file is usually to use messages to a pathname object that represents the generic pathname of a file. Note that there are other properties there, too.

Here the attribute names with standard meanings:

Mode	The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are Lisp and Text.
Package	This attribute specifies the package in which symbols in the file should be interned. The attribute may be either the name of a package, or a list that specifies both the package name and how to create the package if it does not exist. If it is a list, it should look like (*name superpackage initial-size ...options...*). See chapter 27, page 636 for more information about packages.
Base	The number base in which the file is written (remember, it is always parsed in decimal). This affects both *read-base* and *print-base*, since it is confusing to have the input and output bases be different. The most common values are 8 and 10.

Readtable The value specifies the syntax (that is, the choice of readtable) to use for reading Lisp objects from this file. The defined values are **t** or **traditional** for traditional Lisp Machine syntax, and **cl** or **common-lisp** for Common Lisp syntax. If you do not specify this option, the objects in the file are read using whatever readtable is current in the program that reads them.

Lowercase If the attribute value is not **nil**, the file is written in lower-case letters and the editor does not translate to upper case. (The editor does not translate to upper case by default unless the user enables Electric Shift Lock mode.)

Fonts The attribute value is a list of font names, separated by commas. The editor uses this for files that are to be displayed in a specific font, or contain multiple fonts. If this attribute is present, the file is actually stored in the file system with font-change indicators. A font-change indicator is an epsilon (ε) followed by a digit or *. ε*n* means to enter font *n*. The previous font is saved on a stack and ε* means to pop the stack, returning to the previous font. If the file includes an epsilon as part of its contents, it is stored as εε.

When expressions are read from such files, font-change indicators are ignored, and εε is treated as a single ε.

Backspace If the attribute value is not **nil**, **Overstrike** characters in the file should cause characters to overprint on each other. The default is to disallow overprinting and display **Overstrike** the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.

Patch-File If the attribute value is not **nil**, the file is a *patch file*. When it is loaded the system will not complain about function redefinitions. In a patch file, the **defvar** special-form turns into **defconst**; thus patch files always reinitialize variables. Patch files are usually created by special editor commands described in section 28.8, page 672.

Cold-Load A non-**nil** value for this attribute identifies files that are part of the cold load, the core from which a new system version is built. Certain features that do not work in the cold load check this flag to give an error or a compiler warning if used in such files, so that the problem can be detected sooner.

You are free to define additional file attributes of your own. However, to avoid accidental name conflicts, you should choose names that are different from all the names above, and from any names likely to be defined by anybody else's programs.

The following functions are used to examine file attribute lists:

fs:file-attribute-list *pathname*
Returns the attribute list of the file specified by the pathname. This works on both text files and QFASL files.

fs:extract-attribute-list *stream*

Returns the attribute list read from the specified stream, which should be pointing to the beginning of a file. This works on both text streams and QFASL file binary streams. After the attribute list is read, the stream's pointer is set back to the beginning of the file using the :set-pointer file stream operation (see page 468).

fs:read-attribute-list *pathname stream*

pathname should be a pathname object (*not* a string or namelist, but an actual pathname); usually it is a generic pathname (see section 24.5, page 561). *stream* should be a stream that has been opened and is pointing to the beginning of the file whose file attribute list is to be parsed. The attribute list is read from the stream and then corresponding properties are placed on the specified *pathname*. The attribute list is also returned.

The fundamental way that programs in the Lisp Machine notice the presence of properties on a file's attribute list is by examining the property list in the generic pathname. However, there is another way that is more convenient for some applications. File attributes can cause special variables to be bound whenever Lisp expressions are being read from the file—when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base attributes work. You can also deal with attributes this way, by using the following function:

fs:file-attribute-bindings *pathname*

Returns values describing the special variables that should be bound before reading expressions from file *pathname*. It examines the property list of *pathname* and finds all those property names that have fs:file-attribute-bindings properties. Each such property name specifies a set of variables to bind and a set of values to which to bind them. This function returns two values, a list of all the variables and a list of all the corresponding values. Usually you use this function by calling it on a generic pathname that has had fs:read-attribute-list done on it, and then you use the two returned values as the first two arguments of a progv special form (see page 32). Inside the body of the progv the specified bindings will be in effect.

pathname may be anything acceptable as the first argument of get. Usually it is a generic pathname.

Of the standard attribute names, the following ones have fs:file-attribute-bindings, with the following effects. Package binds the variable package (see page 637) to the package. Base binds the variables *print-base* (see page 514) and *read-base* (see page 517) to the value. Readtable binds the variable readtable to a value computed from the specified attribute. Patch-file binds fs:this-is-a-patch-file to the value. Cold-load binds si:file-in-cold-load to the value. Fonts binds si:read-discard-font-changes to t.

Any properties whose names do not have fs:file-attribute-bindings properties are ignored completely.

You can also add your own attribute names that affect bindings. If an indicator symbol has an fs:file-attribute-bindings property, the value of that property is a function that is called when a file with a file attribute of that name is going to be read from. The function is given three arguments: the file pathname, the attribute name, and the attribute value. It must return two values: a list of variables to be bound and a list of values to bind them to. The function for the Base keyword could have been defined by:

```
(defun (:base file-attribute-bindings) (file ignore bse)
  (if (not (and (typep bse 'fixnum)
                (> bse 1)
                (< bse 37.)))
      (ferror 'fs:invalid-file-attrbute
              "File ~A has an illegal -*- Base:~D -*-"
              file bse))
  (values (list 'base 'ibase) (list bse bse)))
```

fs:extract-attribute-bindings *stream*

Returns two values: a list of variables, and a corresponding list of values to bind them to, giving the attribute bindings of the attribute list found on *stream*

fs:invalid-file-attribute (error) *Condition*

An attribute in the file attribute list had a bad value. This is detected within **fs:file-attribute-bindings**.

25.6 Accessing Directories

To understand the functions in this section, it is vital to have read the chapter on *pathnames*. The *filespec* argument in many of these functions may be a pathname or a namestring; its name, type and version default to :wild.

listf *filespec*

Prints on *standard-output* the names of the files that match *filespec*, and their sizes, creation dates, and other information that comes in the directory listing.

fs:directory-list *filespec* &rest *options*

Finds all the files that match *filespec* and returns a list with one element for each file. Each element is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a disembodied property list and **get** may be used to access the file's properties. The car of one element is nil; the properties in this element are properties of the file system as a whole rather than of a specific file.

filespec normally contains wildcards, and the data returned describe all existing files that match it. If it contains no wildcards, it specifies a single file and only that file is described in the data that are returned.

The *options* are keywords which modify the operation. The following options are currently defined:

:noerror	If a file-system error (such as no such directory) occurs during the operation, normally an error is signaled and the user is asked to supply a new pathname. However, if :noerror is specified then, in the event of an error, a condition object describing the error is returned as the result of fs:directory-list. This is identical to the :noerror option to **open**.

:deleted	This is for file servers on which deletion is not permanent. It specifies that deleted (but not yet expunged) files are to be included in the directory listing.
:sorted	This requests that the directory list be sorted by filenames before it is returned.

The properties that may appear in the list of property lists returned by **fs:directory-list** are host-dependent to some extent. The following properties are those that are defined for both ITS and TOPS-20 file servers. This set of properties is likely to be extended or changed in the future.

:length-in-bytes	The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file).
:byte-size	The number of bits in one of those units.
:length-in-blocks	The length of the file in terms of the file system's unit of storage allocation.
:block-size	The number of bits in one of those units.
:creation-date	The date the file was created, as a universal time. See chapter 34, page 776.
:reference-date	The most recent date on which the file was used, as a universal time or nil, meaning the file was never referenced.
:modification-date	The most recent date on which the file's contents were changed, as a universal time.
:author	The name of the person who created the file, as a string.
:reader	The name of the person who last read the file, as a string.
:not-backed-up	t if the file exists only on disk, nil if it has been backed up on magnetic tape.
:directory	t if this file is actually a directory.
:temporary	t if this file is temporary.
:deleted	t if this file is deleted. Deleted files are included in the directory list only if you specify the :deleted option.
:dont-delete	t indicates that the file is not allowed to be deleted.
:dont-supersede	t indicates that the file may not be superseded; that is, a file with the same name and higher version may not be created.
:dont-reap	t indicates that this file is not supposed to be deleted automatically for lack of use.

:dont-dump	t indicates that this file is not supposed to be dumped onto magnetic tape for backup purposes.
:characters	t indicates that this file contains characters (that is, text). **nil** indicates that the file contains binary data. This property, rather than the file's byte size, should be used to decide whether it is a text file.
:link-to	If the file is a link, this property is a string containing the name that the link points to.
:offline	T if the file's contents are not online.
:incremental-dump-date	The last time this file was dumped during an incremental dump (a universal time).
:incremental-dump-tape	The tape on which the last was saved in that incremental dump (a string).
:complete-dump-date	The last time this file was dumped during an full dump (a universal time).
:complete-dump-tape	The tape on which the last was saved in that full dump (a string).
:generation-retention-count	The number of files differing in version that are kept around.
:default-generation-retention-count	The generation-retention-count that a file ordinarily gets when it is created in this directory.
:auto-expunge-interval	The interval at which files are expunged from this directory, in seconds.
:date-last-expunged	The last (universal) time this directory was expunged, or **nil**.
:account	The account to which the file belongs, a string.
:protection	A system-dependent description of the protection of this file as a string.
:physical-volume	A string naming the physical volume on which the file is found.
:volume-name	A string naming the logical volume on which the file is found.
:pack-number	A string describing the pack on which this file is found.
:disk-space-description	A system-dependent description of the space usage on the file system. This usually appears in the plist that applies to the entire directory list.

The element in the directory list that has **nil** instead of a file's pathname describes the directory as a whole.

:physical-volume-free-blocks
: This property is an alist in which each element maps a physical volume name (a string) into a number, that is the number of free blocks on that volume.

:settable-properties
: This property is a list of file property names that may be set. This information is provided in the directory list because it is different for different file systems.

:pathname
: This property is the pathname from which this directory list was made.

:block-size
: This is the number of words in a block in this directory. It can be used to interpret the numbers of free blocks.

fs:directory-list-stream *filespec* &rest *options*

This is like fs:directory-list but returns the information in a different form. Instead of returning the directory list all at once, it returns a special kind of stream which gives out one element of the directory list at a time.

The directory list stream supports two operations: :entry and :close. :entry asks for the next element of the directory stream. :close closes any connection to a remote file server.

The purpose of using fs:directory-list-stream instead of fs:directory-list is that, when communicating with a remote file server, the directory list stream can give you some of the information without waiting for it to all be transmitted and parsed. This is desirable if the directory is being printed on the console.

directory *filespec*

Returns a list of pathnames (truenames) of the files in the directory specified by *filespec*. Wildcards are allowed. This is the Common Lisp way to find the contents of a directory.

fs:expunge-directory *filespec* &key (*error* t)

Expunges the directory specified in *filespec*; that is, permanently eliminates any deleted files in that directory. If *error* is nil, there is no error if the directory does not exist.

Note that not all file systems support this function. To find out whether a particular one does, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion (and therefore expunging).

fs:create-directory *filespec* &key (*error* t)

Creates the directory specified in *filespec*. If *error* is nil, there is no error if the directory cannot be created; instead an error string is returned. Not all file servers support creation of directories.

fs:remote-connect *filespec* &key (*error* t) *access*

Performs the TOPS-20 "connect" or "access" function, or their equivalents, in a remote file server. Access is done if *access* is non-nil; otherwise, connect is done.

The connect operation grants you full access to the specified directory. The access operation grants you whatever access to all files and directories you would have if logged in on the specified directory. Both operations affect access only, since the connected directory of the remote server is never used by the Lisp Machine in choosing which file to operate on.

This function may ask you for a password if one is required for the directory you specify. If the operation cannot be performed, then if *error* is **nil**, an error object is returned.

File Properties:

fs:change-file-properties *file error-p* &rest *properties*

Changes one or more properties of the file *file*. The *properties* arguments are alternating keywords and values. If an error occurs accessing the file or changing the properties, the *error-p* argument controls what is done; if it is **nil**, a condition object describing the error is returned; if it is **t** a Lisp error is signaled. If no error occurs, **fs:change-file-properties** returns **t**.

Only some of the properties of a file may be changed; for instance, its creation date or its author. Exactly which properties may be changed depends on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list** as explained above.

fs:file-properties *file* &optional (*error-p* **t**)

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. The *error-p* argument is the same as in **fs:change-file-properties**.

Filename Completion:

fs:complete-pathname *defaults string type version* &rest *options*

string is a partially-specified file name. (Presumably it was typed in by a user and terminated with the **Altmode** key or the **End** key to request completion.) **fs:complete-pathname** looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded out in a host-dependent fashion.

defaults, *type*, and *version* are the arguments to be given to **fs:merge-pathname-defaults** (see page 558) when the user's input is eventually parsed and defaulted.

options are keywords (without following values) that control how the completion is performed. The following option keywords are allowed:

:deleted	Looks for files which have been deleted but not yet expunged.
:read or **:in**	The file is going to be read. This is the default.
:print or **:write** or **:out**	The file is going to be written (i.e. a new version is going to be created).

:old	Looks only for files that already exist. This is the default.
:new-ok	Allows either a file that already exists or a file that does not yet exist. An example of the use of this is the C-X C-F (Find File) command in the editor.

The first value returned is always a string containing a file name, either the original string or a new, more specific string. The second value returned indicates the success or failure of the completion. It is **nil** if an error occurred. One possible error is that the file is on a file system that does not support completion, in which case the original string is returned unchanged. Other possible second values are :old, which means that the string completed to the name of a file that exists, :new, which means that the string completed to the name of a file that could be created, and **nil** again, which means that there is no possible completion.

Balance Directories:

fs:balance-directories *filespec1 filespec2* &rest *options*

fs:balance-directories is a function for maintaining multiple copies of a directory. Often it is useful to maintain copies of your files on more than one machine; this function provides a simple way of keeping those copies up to date.

The function first parses *filespec1*, filling in missing components with wildcards (except for the version, which is :newest). Then *filespec2* is parsed with *filespec1* as the default. The resulting pathnames are used to generate directory lists using fs:directory-list. Note that the resulting directory lists need not be entire directories; any subset of a directory that fs:directory-list can produce will do.

First the directory lists are matched up on the basis of file name and type. All of the files in either directory list which have both the same name and the same type are grouped together.

The directory lists are next analyzed to determine if the directories are consistent, meaning that two files with the same name and type have equal creation-dates when their versions match, and greater versions have later creation-dates. If any inconsistencies are found, a warning message is printed on the console.

If the version specified for both *filespec1* and *filespec2* was :newest (the default), then the newest version of each file in each directory is copied to the other directory if it is not already there. The result is that each directory has the newest copy of every file in either of the two directories.

If one or both of the specified versions is not :newest, then *every* version that appears in one directory list and not in the other is copied. This has the result that the two directories are completely the same. (Note that this is probably not the right thing to use to *copy* an entire directory. Use copy-file with a wildcard argument instead.)

The *options* are keywords arguments which modify the operation. The following options are currently defined:

:ignore	This option takes one argument, which is a list of file names to ignore when making the directory lists. The default value is **nil**.
:error	This option is identical to the :error option to **open**.
:query-mode	This option takes one argument, which indicates whether or not the user should be asked before files are transferred. If the argument is **nil**, no querying is done. If it is :1->2, then only files being transferred from *filespec2* to *filespec1* are queried, while if it is :2->1, then files transferred from *filespec1* to *filespec2* are queried. If the argument is :always, then the user is asked about all files.
:copy-mode	This option is identical to the :copy-mode option of **copy-file**, and is used to control whether files are treated as binary or textual data.
:direction	This option specifies transfer of files in one direction only. If the value is :1->2 then files are transfered only from *filespec1* to *filespec2*, never in the other direction. If the value is :2->1 then files are transferred only from *filespec2* to *filespec1*. nil, the default, means transfer in either direction as appropriate.

25.7 Errors in Accessing Files

fs:file-error (error) *Condition Flavor*

This flavor is the basis for all errors signaled by the file system.

It defines two special operations, :pathname and :operation. Usually, these return the pathname of the file being operated on, and the operation used. This operation was performed either on the pathname object itself, or on a stream.

It defines prompting for the proceed types :retry-file-operation and :new-pathname, both of which are provided for many file errors. :retry-file-operation tries the operation again exactly as it was requested by the program; :new-pathname expects on argument, a pathname, and tries the same operation on this pathname instead of the original one.

fs:file-operation-failure (fs:file-error) *Condition*

This condition name signifies a problem with the file operation requested. It is an alternative to fs:file-request-failure (page 609), which means that the file system was unable to consider the operation properly.

All the following conditions in this section are always accompanied by **fs:file-operation-failure**, fs:file-error, and error, so they will not be mentioned.

fs:file-open-for-output *Condition*

The request cannot be performed because the file is open for output.

fs:file-locked *Condition*

The file cannot be accessed because it is already being accessed. Just which kinds of simultaneous access are allowed depends on the file system.

fs:circular-link *Condition*

A link could not be opened because it pointed, directly or indirectly through other links, to itself. In fact, some systems report this condition whenever a chain of links exceeds a fixed length.

fs:invalid-byte-size *Condition*

In open, the specified byte size was not valid for the particular file server or file.

fs:no-more-room *Condition*

Processing a request requires resources not available, such as space in a directory, or free disk blocks.

fs:filepos-out-of-range *Condition*

The :set-pointer operation was used with a pointer value outside the bounds of the file.

fs:not-available *Condition*

A requested pack, file, etc. exists but is currently off line or not available to users.

fs:file-lookup-error *Condition*

This condition name categorizes all sorts of failure to find a specified file, for any operation.

fs:device-not-found (fs:file-lookup-error) *Condition*

The specified device does not exist.

fs:directory-not-found (fs:file-lookup-error) *Condition*

The specified directory does not exist.

fs:file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name, type and version. This implies that the device and directory do exist, or one of the errors described above would have been signaled.

fs:multiple-file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name and any of the specified types, in with-open-file-search. Three special operations are defined:

:operation	Returns the function which used with-open-file-search, such as load.
:pathname	The base pathname used.
:pathnames	A list of all the pathnames that were looked for.

fs:link-target-not-found (fs:file-lookup-error) *Condition*

The file specified was a link, but the link's target filename fails to be found.

fs:access-error *Condition*
The operation is possible, but the file server is insubordinate and refuses to obey you.

fs:incorrect-access-to-file (access-error). *Condition*
fs:incorrect-access-to-directory (access-error). *Condition*
The file server refuses to obey you because of protection attached to the file (or, the directory).

fs:invalid-wildcard *Condition*
A pathname had a wildcard in a place where the particular file server does not support them. Such pathnames are not created by pathname parsing, but they can be created with the :new-pathname operation.

fs:wildcard-not-allowed *Condition*
A pathname with a wildcard was used in an operation that does not support it. For example, opening a file with a wildcard in its name.

fs:wrong-kind-of-file *Condition*
An operation was done on the wrong kind of file. If files and directories share one name space and it is an error to open a directory, the error possesses this condition name.

fs:creation-failure *Condition*
An attempt to create a file or directory failed for a reason specifically connected with creation.

fs:file-already-exists (fs:creation-failure) *Condition*
The file or directory to be created already exists.

fs:superior-not-directory (fs:creation-failure fs:wrong-kind-of-file) *Condition*
In file systems where directories and files share one name space, this error results from an attempt to create a file using a filename specifying a directory whose name exists in the file system but is not a directory.

fs:delete-failure *Condition*
A file to be deleted exists, but for some reason cannot be deleted.

fs:directory-not-empty (fs:delete-failure) *Condition*
A file could not be deleted because it is a directory and has files in it.

fs:dont-delete-flag-set (fs:delete-failure) *Condition*
A file could not be deleted because its "don't delete" flag is set.

fs:rename-failure *Condition*
A file to be renamed exists, but the renaming could not be done. The :new-pathname operation on the condition instance returns the specified new pathname, which may be a pathname or a string.

fs:rename-to-existing-file (fs:rename-failure) *Condition*
Renaming cannot be done because there is already a file with the specified new name.

fs:rename-across-directories (fs:rename-failure) *Condition*
Renaming cannot be done because the new pathname contains a different device or directory from the one the file is on. This may not always be an error—some file systems support it in certain cases—but when it is an error, it has this condition name.

fs:unknown-property (fs:change-property-failure) *Condition*
A property name specified in a :change-properties operation is not supported by the file server. (Some file servers support only a fixed set of property names.) The :property operation on the condition instance returns the problematical property name.

fs:invalid-property-value (fs:change-property-failure) *Condition*
In a :change-properties operation, some property was given a value that is not valid for it. The :property operation on the condition instance returns the property name, and the :value operation returns the specified value.

fs:invalid-property-name (fs:change-property-failure) *Condition*
In a :change-properties operation, a syntactically invalid property name was specified. This may be because it is too long to be stored. The :property operation on the condition instance returns the property name.

25.8 File Servers

Files on remote file servers are accessed using *file servers* over the Chaosnet. Normally connections to servers are established automatically when you try to use them, but there are a few ways you can interact with them explicitly.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Lisp Machine characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding; the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, is not the same measured in original Lisp Machine characters as it is measured in the encoded ASCII characters. In the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (Delete), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a quoting character for these codes. Codes 210 (Overstrike), 211 (Tab), 212 (Line), and 214 (Page), are converted to their ASCII cognates, namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (Return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

When a file server is first created for you on a particular host, you must tell the server how to log in on that host. This involves specifying a *username*, and, if the obstructionists are in control of your site, a password. The Lisp Machine prompts you for these on the terminal when they are needed.

Logging in a file server is not the same thing as logging in on the Lisp Machine (see **login**, page 801). The latter identifies you as a user in general and involves specifying one host, your login host. The former identifies you to a particular file server host and must be done for each host on which you access files. However, logging in on the Lisp Machine does specify the username for your login host and logs in a file server there.

The Lisp Machine uses your username (or the part that follows the last period) as a first guess for your password (this happens to take no extra time). If that does not work, you are asked to type a password, or else a username and a password, on the keyboard. You do not have to give the same user name that you are logged in as, since you may have or use different user names on different machines.

Once a password is recorded for one host, the system uses that password as the guess if you connect to a file server on another host.

fs:user-unames *Variable*

This is an alist matching host names with the usernames you have specified on those hosts. Each element is the cons of a host object and the username, as a string.

For hosts running ITS, the symbol **fs:its** is used instead of a host object. This is because every user has the same username on all ITS hosts.

fs:user-host-password-alist *Variable*

Once you have specified a password for a given username and host, it is remembered for the duration of the session in this variable. The value is a list of elements, each of the form

((*username hostname*) *password*)

All three data are strings.

The remembered passwords are used if more than one file server is needed on the same host, or if the connection is broken and a new file server needs to be created.

If you are very scared of your password being known, you can turn off the recording by setting this variable:

fs:record-passwords-flag *Variable*

Passwords are recorded when typed in if this variable is non-**nil**.

You should set the variable at the front of your init file, and also set **fs:user-host-password-alist** to **nil**, since it will already have recorded your password when you logged in.

If you do not use a file server for a period of time, it is killed to save resources on the server host.

fs:host-unit-lifetime *Variable*

This is the length of time after which an idle file server connection should be closed, in 60ths of a second. The default is 20 minutes.

Some hosts have a caste system in which all users are not equal. It is sometimes necessary to enable one's privileges in order to exercise them. This is done with these functions:

fs:enable-capabilities *host* &rest *capabilities*

Enables the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is nil, a default list of capabilities is enabled; the default is also dependent on the operating system type.

fs:disable-capabilities *host* &rest *capabilities*

Disables the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is nil, a default list of capabilities is disabled; the default is also dependent on the operating system type.

The PEEK utility has a mode that displays the status of all your file connections, and of the *host unit* data structures that record them. Clicking on a connection with the mouse gets a menu of operations, of which the most interesting is reset. Resetting a host unit may be useful if the connection becomes hung.

25.8.1 Errors in Communication with File Servers

fs:file-request-failure (fs:file-error error) *Condition*

This condition name categorizes errors that prevent the file system from processing the request made by the program.

The following condition names are always accompanied by the more general classifications fs:file-request-failure, fs:file-error, and error.

fs:data-error *Condition*

This condition signifies inconsistent data found in the file system, indicating a failure in the file system software or hardware.

fs:host-not-available *Condition*

This condition signifies that the file server host is up, but refusing connections for file servers.

fs:network-lossage *Condition*

This condition signifies certain problems in the use of the Chaosnet by a file server, such as failure to open a data connection when it is expected.

fs:not-enough-resources *Condition*

This condition signifies a shortage of resources needed to consider processing a request, as opposed to resources used up by the request itself. This may include running out of network connections or job slots on the server host. It does not include running out of space in a directory or running out of disk space, because these are resources whose requirements come from processing the request.

fs:unknown-operation *Condition*

This condition signifies that the particular file system fails to implement a standardly defined operation; such as, expunging or undeletion on ITS.

26. The Chaosnet

The purpose of the basic software protocol of Chaosnet is to allow high-speed communication among processes on different machines, with no undetected transmission errors.

26.1 Chaosnet Overview

The principal service provided by Chaosnet is a *connection* between two user processes. This is a full-duplex reliable packet-transmission channel. The network undertakes never to garble, lose, duplicate, or resequence the packets; in the event of a serious error it may break the connection off entirely, informing both user processes. User programs may deal explicitly in terms of packets. They may also ignore packet boundaries and treat the connection as two unidirectional streams of 8-bit or 16-bit bytes, but this really works by means of packets.

If you just want to ask a question of another process or host and receive a reply, you can use a *simple transaction*: You send only one packet to the other host, and it sends one packet back. This is more efficient than establishing a connection and using it only briefly. In a simple transaction, the server cannot tell whether the user received the answer; and if the user does not receive the answer, it cannot tell whether the server received the question. In fact, the server might receive the question more than once. If this is unacceptable, a connection must be used.

Each node (or host) on the network is identified by an *address*, which is a 16-bit number. These addresses are used in the routing of packets. There is a table (the system host table, SYS: CHAOS; HOSTS TXT) that relates symbolic host names to numeric host addresses. The host table can record addresses on any number of different networks, and in certain contexts a host address is meaningful only together with the name of the network it is for.

The data transmitted over connections are in units called *packets*. Each packet contains an 8-bit number, the *opcode*, which indicates what its function is. Opcode numbers are always given in octal. Opcodes less than 200 (octal) are special purpose. Each such opcode that is used has an assigned name and a specific function. Users need not know about all of them. Opcodes 200 through 277 (octal) are used for 8-bit user data. Opcodes 300 through 377 (octal) are used for 16-bit user data.

Each packet also contains some number of data bytes, whose meaning depends on the opcode. If the opcode is for user data, then it is up to the application user software to decide on the interpretation.

Establishing a connection:

A connection is created because one process sends a request to a host. The request is a packet containing the special-purpose opcode RFC. The data contains a *contact name* which is used to find the process to connect to. There may be a process on the target host *listening* on this contact name. If so, it decides whether to agree to the connection. Alternatively, the contact name can be the name of a standard service such as `TELNET`. In this case, the receiving host creates a process to respond, loaded with the program for that service.

Once a connection has been established, there is no more need for the contact name and it is discarded. The Lisp Machine remembers what contact name was used to open a connection, but this is only for the user's information.

In the case where two existing processes that already know about each other want to establish a connection, they must agree on a contact name, and then one of them must send the request while the other listens. They must agree between themselves which is to do which.

Contact names are restricted to strings of upper-case letters, numbers, and ASCII punctuation. The maximum length of a contact name is limited only by the packet size, although on ITS hosts the names of automatically-started servers are limited by the file-system to six characters. The contact name is terminated by a space. If the RFC packet contains data beyond the contact name, it is just for interpretation by the listening process, which can also use it in deciding whether to accept the connection.

A simple transaction is also begun with an RFC packet. There is nothing in the RFC packet which indicates whether it is intended to start a connection or a simple transaction. The server has the option of doing either one. But normally any given server always does one or the other, and the requestor knows which one to expect.

The server accepts the request for a connection by sending an OPN packet (a packet with opcode OPN) to the requestor. It can also refuse the connection by sending a CLS packet. The data in the CLS packet is a string explaining the reason for the refusal. Another alternative is to tell the requestor to try a different host or a different contact name. This is called *forwarding* the request, and is done with a FWD packet.

The server can also respond with an answer, an ANS packet, which is the second half of a simple transaction. (Refusing and forwarding are also meaningful when a simple transaction is intended, just as when a connection is intended).

Once the connection is open:

Data transmitted through Chaosnet generally follow Lisp Machine standards. Bits and bytes are numbered from right to left, or least-significant to most-significant. The first 8-bit byte in a 16-bit word is the one in the arithmetically least-significant position. The first 16-bit word in a 32-bit double-word is the one in the arithmetically least-significant position. This is the "little-endian" convention.

Big-endian machines such as the PDP-10 need to reorder the characters in a word in order to access them conveniently. For their sake, some packet opcodes imply 8-bit data and some imply 16-bit data. Packets known to contain 8-bit bytes, including opcodes 200 through 277, are stored in the big-endian machine's memory a character at a time, whereas packets containing 16-bit data are stored 16 bits at a time.

The character set used is dictated by the higher-level protocol in use. Telnet and Supdup, for example, each specifies its own ASCII-based character set. The default character set—used for new protocols and for text that appears in the basic Chaosnet protocol, such as contact names—is the Lisp Machine character set.

If one process tries to send data faster than the other can process it, the buffered packets could devour lots of memory. Preventing this is the purpose of *flow control*. Each process specifies a *window size*, which is the number of packets that are allowed to be waiting for that process to read. Attempting to send on a connection whose other side's window is full waits until the other side reads some packets. The default window size is 13, but for some applications you might wish to specify a larger value (see **chaos:connect**, page 615). There is little reason ever to specify a smaller value.

Breaking a connection:

Either end of a connection can break the connection abruptly by sending a CLS packet. The data in this packet is a string describing why the connection was broken.

To break a connection gently, it is necessary to verify that all the data transmitted was received properly before sending a CLS. This matters in some applications and is unnecessary in others. When it is needed, it is done by sending a special packet, an EOF packet, which is mostly like a data packet except for its significance with regard to closing the connection. The EOF packet is like the words "the end" at the end of a book: it tells the recipient that it has received all the data it is supposed to receive, that there are no missing pages that should have followed. When the sender of the EOF sees the acknowledgement for the EOF packet, indicating that the EOF was received and understood, it can break the connection with a CLS.

If a process that expects to receive an EOF gets a CLS with no EOF, it takes this to mean that the connection was broken before the transmission was finished. If the process does receive an EOF, it does not break the connection itself immediately. It waits to see the sender of the EOF break it. If this does not happen in a few seconds, the EOF recipient can break the connection.

It is illegal to put data in an EOF packet; in other words, the byte count should always be zero. Most Chaosnet implementations simply ignore any data that is present in an EOF.

If both sides are sending data and both need to know for certain where "the end" is, they must do something a little more complicated. Arbitrarily call one party the user and the other the server. The protocol is that after sending all its data, each party sends an EOF and waits for it to be acknowledged. The server, having seen its EOF acknowledged, sends a second EOF. The user, having seen its EOF acknowledged, looks for a second EOF and *then* sends a CLS and goes away. The server goes away when it sees the user's CLS, or after a brief timeout has elapsed. This asymmetrical protocol guarantees that each side gets a chance to know that both sides agree that all the data have been transferred. The first CLS is sent only after both sides have waited for their (first) EOF to be acknowledged.

Clearing up inconsistencies:

If a host crashes, it is supposed to forget all the connections that it had. When a packet arrives on one of the former connections, the host will report "no such connection" to the sender with a LOS packet, whose data is a string explaining what happened. The same thing happens if a CLS packet is lost; the intended recipient may keep trying to use the connection that the other side (which sent the CLS) no longer believes should exist. LOS packets are used whenever a host receives a packet that it should not be getting; the recipient of the LOS packet knows that the connection it thought it was using does not exist any more.

26.2 Conns

On the Lisp Machine, your handle on a connection is a named structure of type chaos:conn. The conn may have an actual connection attached to it, or it may have a connection still being made, or record that a connection was refused, closed or broken.

chaos:inactive-state
: This conn is not really in use at all.

chaos:rfc-sent-state
: This conn was used to request a connection to another process, but no reply has been received. When the reply is received, it may change the conn's state to chaos:answered-state, chaos:cls-received-state, or chaos:open-state.

chaos:listening-state
: This conn is being used to listen with. If a RFC packet is received for the contact name you are listening on, the state changes to chaos:rfc-received-state.

chaos:rfc-received-state
: This means that your listen has "heard" an RFC packet that matches it. You can accept, reject, forward or answer the request. Accepting goes to state chaos:open-state; refusing or forwarding goes to to state chaos:inactive-state.

chaos:open-state
: This conn is one end of an open connection. You can receive any data packets that are waiting and you can transmit data.

chaos:answered-state
: This conn was used to send an RFC packet and an ANS packet was received in response (a simple transaction answer arrived). You can read the ANS packet, that is all.

chaos:cls-received-state
: This conn has received a CLS packet (the connection was closed or refused). You can read any data packets that came in before the CLS; after them you can read the CLS.

chaos:los-received-state
: This conn's connection was broken and the other end sent a LOS packet to say so. The LOS packet is the only packet available to be read.

chaos:host-down-state
: The host at the other end of this conn's connection has not responded to anything for a significant time.

chaos:foreign-state
: The connection is being used with a foreign protocol encapsulated in UNC packets (see the MIT AI Lab memo entitled "Chaosnet" for more information on this).

These are the fields of a conn that you might be interested in:

chaos:conn-state *conn*

This slot holds the state of *conn*. It is one of the symbols listed above.

chaos:conn-foreign-address *conn*

Returns the address of the host at the other end of this connection. Use si:get-host-from-address to find out which host this is (see page 577).

chaos:conn-read-pkts *conn*

Internally threaded chain of incoming packets available to be read from *conn*.

Its main use for the applications programmer is to test whether there are any incoming packets.

chaos:conn-window-available *conn*

Returns the number of packets you may transmit before the network software forces you to wait for the receiver to read some. This is just a minimum. By the time you actually send this many packets, the receiver may already have said he has room for some more.

chaos:conn-plist *conn*

This slot is used to store arbitrary properties on *conn*. You can store properties yourself; use property names that are not in the chaos package to avoid conflict.

chaos:contact-name *conn*

Returns the contact name with which *conn* was created. The contact name is not significant to the functioning of the connection once an RFC and LSN have matched, but it is remembered for the sake of debugging.

chaos:wait *conn state timeout* &optional *whostate*

Waits until the state of *conn* is not the symbol *state*, or until *timeout* 60ths of a second have elapsed. If the timeout occurs, nil is returned; otherwise t is returned. *whostate* is the process state to put in the who-line; it defaults to "Chaosnet wait".

26.3 Opening and Closing Connections

26.3.1 User-Side

chaos:connect *host contact-name* &optional *window-size timeout*

Opens a stream connection; returns a conn if it succeeds or else a string giving the reason for failure. *host* may be a number or the name of a known host. *contact-name* is a string containing the contact name and any additional arguments to go in the RFC packet. If *window-size* is not specified it defaults to 13. If *timeout* is not specified it defaults to 600 (ten seconds).

chaos:simple *host contact-name* &optional *timeout*

Taking arguments similar to those of chaos:connect, this performs the user side of a simple-transaction. The returned value is either an ANS packet or a string containing a failure message. The ANS packet should be disposed of (using chaos:return-pkt, see below) when you are done with it.

chaos:remove-conn *conn*

Makes *conn* null and void. It becomes inactive, all its buffered packets are freed, and the corresponding Chaosnet connection (if any) goes away. This is called *removing* the connection. *conn* itself is marked for reuse for another Chaosnet connection, so you should not do anything else with it after it is removed.

chaos:close-conn *conn* &optional *reason*

Closes and removes the connection. If it is open, a CLS packet is sent containing the string *reason*. Don't use this to reject RFC's; use chaos:reject for that.

chaos:open-foreign-connection *host index* &optional *pkt-allocation distinguished-port*

Creates a conn that may be used to transmit and receive foreign protocols encapsulated in UNC packets. *host* and *index* are the destination address for packets sent with chaos:send-unc-pkt. *pkt-allocation* is the 'window size', i.e. the maximum number of input packets that may be buffered. It defaults to 10. If *distinguished-port* is supplied, the local index is set to it. This is necessary for protocols that define the meanings of particular index numbers.

See the MIT AI Lab memo entitled "Chaosnet" for more information on using foreign protocols.

26.3.2 Server-Side

chaos:listen *contact-name* &optional *window-size wait-for-rfc*

Waits for an RFC for the specified contact name to arrive, then returns a conn that is in chaos:rfc-received-state. If *window-size* is not specified it defaults to 13. If *wait-for-rfc* is specified as nil (it defaults to t) then the conn is returned immediately without waiting for an RFC to arrive.

chaos:server-alist *Variable*

Contains an entry for each server that always exists. When an RFC arrives for one of these servers, the specified form is evaluated in the background process; typically it creates a process that will then do a chaos:listen. Use the add-initialization function to add entries to this list.

chaos:accept *conn*

conn must be in chaos:rfc-received-state. An OPN packet is transmitted and *conn* enters the chaos:open-state. If the RFC packet has not already been read with chaos:get-next-pkt, it is discarded. You should read it before accepting, if it contains arguments in addition to the contact name.

chaos:reject *conn reason*
conn must be in chaos:rfc-received-state. A CLS packet containing the string *reason* is sent and *conn* is removed from the connection table.

chaos:forward-all *contact-name host*
Causes all future requests for connection to this host on *contact-name* to be forwarded to the same contact name at host *host*.

chaos:answer-string *conn string*
conn must be in chaos:rfc-received-state. An ANS packet containing the string *string* is sent and *conn* is removed from the connection table.

chaos:answer *conn pkt*
conn must be in chaos:rfc-received-state. *pkt* is transmitted as an ANS packet and *conn* is removed. Use this function when the answer is some binary data rather than a text string.

chaos:fast-answer-string *contact-name string*
If a pending RFC exists to *contact-name*, an ANS containing *string* is sent in response to it and t is returned. Otherwise nil is returned. This function involves the minimum possible overhead. No conn is created.

26.4 Stream Input and Output

chaos:open-stream *host contact-name* &key *window-size timeout error direction characters ascii-translation*
Opens a Chaosnet connection and returns a stream that does I/O to it. *host* is the host to connect to; *contact-name* is the contact name at that host. These two arguments are passed along to chaos:connect.

If *host* is nil, a connection to *contact-name* is listened for, and a stream is returned as soon as a request comes in for that contact name. At this time, you must accept or reject the connection by invoking the stream operation :accept or :reject. Before you decide which to do, you can use the :foreign-host operation to find out where the connection came from.

The remaining arguments are:

window-size *timeout*	These two arguments specify two arguments for chaos:connect.
error	If the value is non-nil, a failure to connect causes a Lisp error. Otherwise, it causes a string describing the error to be returned.
direction *characters* *ascii-translation*	These three arguments are passed along to chaos:make-stream.

chaos:make-stream *conn* &key *direction characters ascii-translation*
Creates and returns a stream that does I/O on the connection *conn*, which should be open as a stream connection. *direction* may be :input, :output or :bidirectional.

If *characters* is non-nil (which is the default), the stream reads and writes 8-bit bytes. If *characters* is nil, the stream reads and writes 16-bit bytes.

If *ascii-translation* is non-nil, characters written to the stream are translated to standard ASCII before they are sent, and characters read are translated from ASCII to the Lisp Machine character set.

:foreign-host *Operation on* chaos:basic-stream
Returns the host object for the host at the other end of this stream's connection.

:accept *Operation on* chaos:basic-stream
Accepts the request for a connection which this stream received. Used only for streams made by chaos:open-stream with nil as the *host* argument.

:reject *reason-string* *Operation on* chaos:basic-stream
Rejects the request for a connection which this stream received, sending *reason-string* in the CLS packet as the reason. Used only for streams made by chaos:open-stream with nil as the *host* argument.

:close &optional *abort-p* *Operation on* chaos:basic-stream
Sends a CLS packet and removes the connection. For output connections and bidirectional connections, the :eof operation is performed first, if *abort-p* is nil.

:force-output *Operation on* chaos:basic-output-stream
Any buffered output is transmitted. Normally output is accumulated until a full packet's worth of bytes are available, so that maximum-size packets are transmitted.

:finish *Operation on* chaos:basic-output-stream
Waits until either all packets have been sent and acknowledged, or the connection ceases to be open. If successful, returns t; if the connection goes into a bad state, returns nil.

:eof *Operation on* chaos:basic-output-stream
Forces out any buffered output, sends an EOF packet, and does a :finish.

:clear-eof *Operation on* chaos:basic-input-stream
Allows you to read past an EOF packet on input. Normally, each :tyi done at eof returns nil or signals the specified eof error. If you do :clear-eof on the stream, you can then read more data (assuming there are data packets following the EOF packet).

26.5 Packet Input and Output

Input and output on a Chaosnet connection can be done at the whole-packet level, using the functions in this section. A packet is represented by a **chaos:pkt** data structure. Allocation of **pkts** is controlled by the system; each **pkt** that it gives you must be given back. There are functions to convert between **pkts** and strings. A **pkt** is an **art-16b** array containing the packet header and data; the leader of a **pkt** contains a number of fields used by the system.

chaos:first-data-word-in-pkt *Constant*

This is the index in any **pkt** of the element that is the first 16-bit word of user data. (Preceding elements are used to store a header used by the hardware.)

chaos:max-data-words-per-pkt *Constant*

The maximum number of 16-bit data words allowed in a packet.

chaos:pkt-opcode *pkt*

Accessor for the opcode of the packet *pkt*. To set the opcode, do

```
(setf (chaos:pkt-opcode my-pkt) my-opcode)
```

The system provides names for all the opcodes standardly used. The names useful to the applications programmer appear at the end of this section.

chaos:pkt-nbytes *pkt*

Accessor for the number-of-data-bytes field of *pkt*'s. This field says how much of *pkt*'s contects are valid data, measured in 8-bit bytes. This field can be set with **setf** also.

chaos:pkt-string *pkt*

An indirect array that is the data field of *pkt* as a string of 8-bit bytes. The length of this string is equal to (**chaos:pkt-nbytes** *pkt*). If you wish to record the contects of *pkt* permanently, you must copy this string.

chaos:set-pkt-string *pkt* &rest *strings*

Copies the *strings* into the data field of *pkt*, concatenating them, and sets (**chaos:pkt-nbytes** *pkt*) accordingly.

chaos:get-pkt

Allocates a **pkt** for use by the user.

chaos:return-pkt *pkt*

Returns *pkt* to the system for reuse. The packets given to you by **chaos:get-pkt**, **chaos:get-next-pkt** and **chaos:simple** should be returned to the system in this way when you are finished with them.

chaos:send-pkt *conn pkt* &optional (*opcode* **chaos:dat-op**)

Transmits *pkt* on *conn*. *pkt* should have been allocated with **chaos:get-pkt** and then had its data field and n-bytes filled in. *opcode* must be a data opcode (#o200 or more) or EOF. An error is signaled, with condition **chaos:not-open-state**, if *conn* is not open.

Giving a **pkt** to **chaos:send-pkt** constitutes giving it back to the system. You do not need to call **chaos:return-pkt**.

chaos:send-string *conn* &rest *strings*
Sends a data packet containing the concatenation of *strings* as its data.

chaos:send-unc-pkt *conn pkt* &optional *pkt-number ack-number*
Transmits *pkt*, an UNC packet, on *conn*. The opcode, packet number, and acknowledge number fields in the packet header are filled in (the latter two only if the optional arguments are supplied).

See the MIT AI Lab memo entitled "Chaosnet" for more information on using foreign protocols.

chaos:may-transmit *conn*
A predicate that returns t if there is any space in the window for transmitting on *conn*. If the value is nil, you may have to wait if you try to transmit. If the value is t, you certainly do not have to wait.

chaos:finish-conn *conn* &optional (*whostate* "Net Finish")
Waits until either all packets have been sent and acknowledged, or the connection ceases to be open. If successful, returns t; if the connection goes into a bad state, returns nil. *whostate* is the process state to display in the who-line while waiting.

chaos:conn-finished-p *conn*
t unless *conn* is open and has sent packets which have not been acknowledged.

chaos:get-next-pkt *conn* &optional (*no-hang-p* nil) *whostate check-conn-state*
Returns the next input packet from *conn*. When you are done with the packet you must give it back to the system with chaos:return-pkt. This can return an RFC, CLS, or ANS packet, in addition to data, UNC, or EOF.

If no packets are available, nil is returned if *no-hang-p* is t. Otherwise, chaos:get-next-pkt waits for a packet to come in or for the state to change. *whostate* is displayed in the who line; it defaults to "Chaosnet Input".

If *check-conn-state* is non-nil, the connection state is checked for validity before anything else is done, and an error is signaled if the connection is in a bad state, with condition name chaos:host-down, chaos:los-received-state, or chaos:read-on-closed-connection. If *check-conn-state* is nil and *no-hang-p* is t, nil is returned. *check-conn-state* defaults to (not *no-hang-p*).

chaos:data-available *conn*
A predicate that returns t if there any input packets available from *conn*.

Here are symbolic names for the opcodes that an applications programmer needs to know about:

chaos:rfc-op *Constant*

This special-purpose opcode is used for requesting a connection. The data consists of the contact name terminated by a space character, followed optionally by additional data whose meaning is up to the server for that contact name.

chaos:lsn-op *Constant*

This special-purpose opcode is used when you ask to listen on a contact name. The data is just the contact name. This packet is never actually sent over the network, just kept in the Chaosnet software and compared with the contact names in RFC packets that arrive.

chaos:opn-op *Constant*

This special-purpose opcode is used by the server process to accept the request for a connection conveyed by an RFC packet. Its data serves only internal functions.

chaos:ans-op *Constant*

This special-purpose opcode is used to send a simple reply. The simple reply is sent back in place of opening a connection.

chaos:los-op *Constant*

This special-purpose packet is what you receive if you try to use a connection that has been broken. Its data is a message explaining the situation, which you can print for the user.

chaos:cls-op *Constant*

This special-purpose packet is used to close a connection. Its data is a message explaining the reason, and it can be printed for the user. Note that you cannot count on receiving a CLS packet because it is not retransmitted if it is lost. If that happens you get a LOS when you try to use the connection (thinking it is still open).

CLS packets are also used for refusing to open a connection in the first place.

chaos:eof-op *Constant*

This special-purpose opcode is used to indicate the end of the data that you really want to transmit. When this packet is acknowledged by the other process, you know that all the real data was received properly. You can wait for this with chaos:finish. The EOF packet carries no data itself.

chaos:dat-op *Constant*

This is opcode 200 (octal), which is the normal opcode used for 8-bit user data. Some protocols use multiple data opcodes in the range 200 through 277, but simple protocols that do not need to distinguish types of packets just use opcode 200.

26.6 Connection Interrupts

chaos:interrupt-function *conn*

This attribute of a **conn** is a function to be called when certain events occur on this connection. Normally this is **nil**, which means not to call any function, but you can use **setf** to store a function here. Since the function is called in the Chaosnet background process, it should not do any operations that might have to wait for the network, since that could permanently hang the background process.

The function's first argument is one of the following symbols, giving the reason for the interrupt. The function's second argument is *conn*. Additional arguments may be present depending on the reason. The possible reasons are:

:input	A packet has arrived for the connection when it had no input packets queued. It is now possible to do **chaos:get-next-pkt** without having to wait. There are no additional arguments.
:output	An acknowledgement has arrived for the connection and made space in the window when formerly it was full. Additional output packets may now be transmitted with **chaos:send-pkt** without having to wait. There are no additional arguments.
:change-of-state	The state of the connection has changed. The third argument to the function is the symbol for the new state.

chaos:read-pkts *conn*

Some interrupt functions want to look at the queued input packets of a connection when they get a :input interrupt. **chaos:read-pkts** returns the first packet available for reading. Successive packets can be found by following **chaos:pkt-link**.

chaos:pkt-link *pkt*

Lists of packets in the NCP are threaded together by storing each packet in the chaos:pkt-link of its predecessor. The list is terminated with **nil**.

26.7 Chaosnet Errors

sys:network-error (error) *Condition Flavor*

All errors from the Chaosnet code use flavors built on this one.

26.7.1 Local Problems

sys:local-network-error (sys:network-error error) *Condition Flavor*

This flavor is used for problems in connection with the Chaosnet that have entirely to do with what is going on in this Lisp Machine.

sys:network-resources-exhausted *Condition*
(sys:local-network-error sys:network-error error)

Signaled when some local resource in the NCP was exhausted. Most likely, there are too many Chaosnet connections and the connection table is full.

sys:unknown-address (sys:local-network-error sys:network-error error) *Condition*

The *address* argument to chaos:connect or some similar function was not recognizable. The :address operation on the condition instance returns the address that was supplied.

26.7.2 Problems Involving Other Machines' Actions

sys:remote-network-error (sys:network-error error) *Condition Flavor*

This flavor is used for network problems that involve the actions (or lack of them) of other machines. It is often useful to test for as a condition name.

The operations :connection and :foreign-host return the chaos:conn object and the host object for the foreign host.

All the condition names listed below imply the presence of sys:remote-network-error, sys:network-error and error. For brevity, these are not mentioned in the individual descriptions.

Every instance of sys:remote-network-error is either a sys:connection-error or a sys:bad-connection-state.

sys:connection-error *Condition*

This condition name categorizes failure to complete a connection.

sys:bad-connection-state *Condition*

This condition name categorizes errors where an existing, valid connection becomes invalid. The error is not signaled until you try to use the connection.

sys:host-not-responding *Condition*

This condition name categorizes errors where no packets whatever are received from the foreign host, making it seem likely that that host or the network is down.

sys:host-not-responding-during-connection *Condition*
(sys:connection-error sys:host-not-responding)

This condition is signaled when a host does not respond while it is being asked to make a connection.

sys:no-server-up (sys:connection-error) *Condition*

This condition is signaled by certain functions which request service from any available machine which can provide it, if no such machine is responding.

sys:host-stopped-responding (sys:bad-connection-state sys:host-not-responding) *Condition*

This condition is signaled when a host does not respond even though a connection to it already exists.

sys:connection-refused (sys:connection-error) *Condition*

This is signaled when a connection is refused.

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

sys:connection-closed (sys:bad-connection-state) *Condition*

This is signaled when you try to send on a connection which has been closed by the other host.

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

sys:connection-lost (sys:bad-connection-state) *Condition*

This is signaled when you try to use a connection on which a LOS packet was received.

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

sys:connection-no-more-data (sys:bad-connection-state) *Condition*

This is signaled when you try to read from a connection which has been closed by the other host, when there are no packets left to be read. (It is no error to read from a connection which has been closed, if you have not yet read all the packets which arrived, including the CLS packet).

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

26.8 Information and Control

chaos:host-up-p *host* &optional (*timeout* 180.)

t if *host* responds over the Chaosnet within *timeout* sixtieths of a second, otherwise nil. The value is always nil if *host* is not on the Chaosnet.

chaos:up-hosts *host-list* &optional *number-of-hosts* (*timeout* 250.)

Returns a list of all the hosts in *host-list* which are currently responding over the Chaosnet. *host-list* is a list of host names and/or host objects. The value is always a list of host objects, possibly nil for none of them.

If *number-of-hosts* is non-nil, it should be a positive integer; when that many hosts have responded, chaos:up-hosts returns right away without bothering to listen for replies from the rest.

timeout is an integer; if a host fails to respond for that many sixtieths of a second, it is assumed to be down.

chaos:host-data &optional *host*
host may be a number or a known host name, and defaults to the local host. Two values are returned. The first value is the host name and the second is the host number. If the host is a number not in the table, it is asked its name using the STATUS protocol; if no response is received the name "Unknown" is returned.

chaos:print-conn *conn* &optional (*short* t)
Prints everything the system knows about the connection. If *short* is nil it also prints everything the system knows about each queued input and output packet on the connection.

chaos:print-pkt *pkt* &optional (*short* nil)
Prints everything the system knows about the packet, except its data field. If *short* is t, only the first line of the information is printed.

chaos:print-all-pkts *pkt* &optional (*short* t)
Calls chaos:print-pkt on *pkt* and all packets on the threaded list emanating from it.

chaos:status
Prints the hardware status.

chaos:reset
Resets the hardware and software and turns off Chaosnet communication.

chaos:assure-enabled
Turns on Chaosnet communication if it is not already on. It is normally always on unless you call one of the functions in this section.

chaos:enable
Resets the hardware and turns on Chaosnet communication.

chaos:disable
Resets the hardware and turns off Chaosnet communication.

chaos:show-routing-table *host* &optional (*stream* *standard-output)
Print out *host*'s routing table onto *stream*.

chaos:show-routing-path &key (*from si:local-host*) *to* (*stream* *standard-output)
Show how a packet would get from *from* to *to*. For this to work when the hosts are on different subnets, the bridge must respond to the DUMP-ROUTING-TABLE request.

The PEEK program has a mode that displays the status of all of the Lisp Machine's Chaosnet connections, and various other information, in a continuously updating fashion.

26.9 Higher-Level Protocols

This section briefly documents some of the higher-level protocols of the most general interest. There are quite a few other protocols which are too specialized to mention here. All protocols other than the STATUS protocol are optional and are only implemented by those hosts that need them. All hosts are required to implement the STATUS protocol since it is used for network maintenance.

The site files tell the Lisp Machine which hosts at your site implement certain higher-level protocols. See section 35.12, page 810.

26.9.1 Status

All network nodes, even bridges, are required to answer RFC's with contact name STATUS, returning an ANS packet in a simple transaction. This protocol is primarily used for network maintenance. The answer to a STATUS request should be generated by the Network Control Program, rather than by starting up a server process, in order to provide rapid response.

The STATUS protocol is used to determine whether a host is up, to determine whether an operable path through the network exists between two hosts, to monitor network error statistics, and to debug new Network Control Programs and new Chaosnet hardware. The **hostat** function on the Lisp Machine uses this protocol.

The first 32 bytes of the ANS contain the name of the node, padded on the right with zero bytes. The rest of the packet contains blocks of information expressed in 16-bit and 32-bit words, low byte first (little-endian convention). The low-order half of a 32-bit word comes first. Since ANS packets contain 8-bit data (not 16-bit), big-endian machines such as PDP-10s have to shuffle the bytes explicitly when using this protocol. The first 16-bit word in a block is its identification. The second 16-bit word is the number of 16-bit words to follow. The remaining words in the block depend on the identification.

This is the only block type currently defined. All items are optional, according to the count field, and extra items not defined here may be present and should be ignored. Note that items after the first two are 32-bit words.

word 0	A number between 400 and 777 octal. This is 400 plus a subnet number. This block contains information on this host's direct connection to that subnet.
word 1	The number of 16-bit words to follow, usually 16.
words 2-3	The number of packets received from this subnet.
words 4-5	The number of packets transmitted to this subnet.
words 6-7	The number of transmissions to this subnet aborted by collisions or because the receiver was busy.

words 8-9	The number of incoming packets from this subnet lost because the host had not yet read a previous packet out of the interface and consequently the interface could not capture the packet.
words 10-11	The number of incoming packets from this subnet with CRC errors. These were either transmitted wrong or damaged in transmission.
words 12-13	The number of incoming packets from this subnet that had no CRC error when received, but did have an error after being read out of the packet buffer. This error indicates either a hardware problem with the packet buffer or an incorrect packet length.
words 14-15	The number of incoming packets from this subnet that were rejected due to incorrect length (typically not a multiple of 16 bits).
words 16-17	The number of incoming packets from this subnet rejected for other reasons (e.g. too short to contain a header, garbage byte-count, forwarded too many times.)

If word 0, the identification, is a number between 0 and 377 octal, this is an obsolete format of block. The identification is a subnet number and the counts are as above except that they are only 16 bits instead of 32, and consequently may overflow. This format should no longer be sent by any hosts.

Identification numbers of 1000 octal and up are reserved for future use.

26.9.2 Routing Information

For network and NCP debugging, this RFC/ANS protocol should be implemented. The contact name is `DUMP-ROUTING-TABLE`, and the response is an ANS packet whose words alternate contain a method to getting to a subnet, and the cost. If the method is zero, then the machine knows of know what to get to that subnet. If the method is positive and less than 400 (octal), it is an interface of some kind to that subnet. If the method is 400 (octal) or greater, this is actually a bridge (host) off which the machine is bouncing packets destined for the subnet.

26.9.3 Telnet and Supdup

The Telnet and Supdup protocols of the Arpanet exist in identical form in Chaosnet. These protocols allow access to a computer system as an interactive terminal from another network node.

The contact names are `TELNET` and `SUPDUP`. The direct borrowing of the Telnet and Supdup protocols was eased by their use of 8-bit byte streams and of only a single connection. Note that these protocols define their own character sets, which differ from each other and from the Chaosnet standard character set.

For the Telnet protocol, refer to the Arpanet Protocol Handbook. For the Supdup protocol, see MIT AI Lab memo 644.

Chaosnet contains no counterpart of the INR/INS attention-getting feature of the Arpanet. The Telnet protocol sends a packet with opcode 201 octal in place of the INS signal. This is a controlled packet and hence does not provide the "out of band" feature of the Arpanet INS, however it is satisfactory for the Telnet 'interrupt process' and 'discard output' operations on the kinds of hosts attached to Chaosnet.

26.9.4 File Access

The FILE protocol is primarily used by Lisp Machines to access files on network file servers. ITS and TOPS-20 are equipped to act as file servers. A user end for the file protocol also exists for TOPS-20 and is used for general-purpose file transfer. For complete documentation on the file protocol, see SYS: DOC; FILE TEXT. The Arpanet file transfer protocols have not been implemented on the Chaosnet (except through the Arpanet gateway described below).

26.9.5 Mail

The MAIL protocol is used to transmit inter-user messages through the Chaosnet. The Arpanet mail protocol was not used because of its complexity and poor state of documentation. This simple protocol is by no means the last word in mail protocols; however, it is adequate for the mail systems we presently possess.

The sender of mail connects to contact name MAIL and establishes a stream connection. It then sends the names of all the recipients to which the mail is to be sent at (or via) the server host. The names are sent one to a line and terminated by a blank line (two carriage returns in a row). The Lisp Machine character set is used. A reply (see below) is immediately returned for each recipient. A recipient is typically just the name of a user, but it can be a user-atsign-host sequence or anything else acceptable to the mail system on the server machine. After sending the recipients, the sender sends the text of the message, terminated by an EOF. After the mail has been successfully swallowed, a reply is sent. After the sender of mail has read the reply, both sides close the connection.

In the MAIL protocol, a reply is a signal from the server to the user (or sender) indicating success or failure. The first character of a reply is a plus sign for success, a minus sign for permanent failure (e.g. no such user exists), or a percent sign for temporary failure (e.g. unable to receive message because disk is full). The rest of a reply is a human-readable character string explaining the situation, followed by a carriage return.

The message text transmitted through the mail protocol normally contains a header formatted in the Arpanet standard fashion. Refer to the Arpanet Protocols Handbook.

26.9.6 Send

The SEND protocol is used to transmit an interactive message (requiring immediate attention) between users. The sender connects to contact name `SEND` at the machine to which the recipient is logged in. The remainder of the RFC packet contains the name of the person being sent to. A stream connection is opened and the message is transmitted, followed by an EOF. Both sides close after following the end-of-data protocol described in page 613. The fact that the RFC was responded to affirmatively indicates that the recipient is in fact present and accepting messages. The message text should begin with a suitable header, naming the user that sent the message. The standard for such headers, not currently adhered to by all hosts, is one line formatted as in the following example:

```
        Moon@MIT-MC 6/15/81 02:20:17
```

Automatic reply to the sender can be implemented by searching for the first '@' and using the SEND protocol to the host following the '@' with the argument preceding it.

26.9.7 Name

The Name/Finger protocol of the Arpanet exists in identical form on the Chaosnet. Both Lisp Machines and timesharing machines support this protocol and provide a display of the user(s) currently logged in to them.

The contact name is `NAME`, which can be followed by a space and a string of arguments like the "command line" of the Arpanet Name protocol. A stream connection is established and the "finger" display is output in Lisp Machine character set, followed by an EOF.

Lisp Machines also support the FINGER protocol, a simple-transaction version of the NAME protocol. An RFC with contact name `FINGER` is transmitted and the response is an ANS containing the following items of information separated by carriage returns: the logged-in user ID, the location of the terminal, the idle time in minutes or hours-colon-minutes, the user's full name, and the user's group affiliation.

26.9.8 Time

The Time protocol allows a host such as a Lisp Machine that has no long-term timebase to ask the time of day. An RFC to contact name `TIME` evokes an ANS containing the universal time as a 32-bit number in four 8-bit bytes, least-significant byte first.

26.9.9 Uptime

This is similar to the TIME protocol, except that the contact name is `UPTIME`, and the time returned is actually an interval (in seconds) describing how long the host has been up.

26.9.10 Arpanet Gateway

This protocol allows a Chaosnet host to access almost any service on the Arpanet. The gateway server runs on each ITS host that is connected to both networks. It creates an Arpanet connection and a Chaosnet connection and forwards data bytes from one to the other. It also provides for a one-way auxiliary connection, used for the data connection of the Arpanet File Transfer Protocol.

The RFC packet contains a contact name of `TCP`, a space, the name of the Arpanet host to be connected to, optionally followed by a space and the contact-socket number in octal, which defaults to 1 if omitted. The name of host can also be an Internet-format address. The bi-directional 8-bit connection is made by connecting to the host with TCP.

If a data packet with opcode 201 (octal) is received, an Arpanet INS signal is transmitted. Any data bytes in this packet are transmitted normally. (This does nothing in the current server, since TCP does not define an interrupt signal.)

If a data packet with opcode 210 (octal) is received, an auxiliary connection on each network is opened. The first eight data bytes are the Chaosnet contact name for the auxiliary connection; the user should send an RFC with this name to the server. The next four data bytes are the TCP socket number to be connected to, in the wrong order, most-significant byte first. The byte-size of the auxiliary connection is 8 bits.

The normal closing of an TCP connection corresponds to an EOF packet. Closing due to an error, such as Host Dead, corresponds to a CLS packet.

26.9.11 Host Table

The `HOSTAB` protocol may be used to access tables of host addresses on other networks, such as the Arpanet or Internet. Servers for this protocol currently exist for Tenex, TOPS-20, ITS, and Lisp Machines.

The user connects to contact name `HOSTAB`, undertakes a number of transactions, then closes the connection. Each transaction is initiated by the user transmitting a host name followed by a carriage return. The server responds with information about that host, terminated with an EOF, and is then ready for another transaction. The server's response consists of a number of attributes of the host. Each attribute consists of an identifying name, a space character, the value of the attribute, and a carriage return. Values may be strings (free of carriage returns and *not* surrounded by double-quotes) or octal numbers. Attribute names and most values are in upper case. There can be more than one attribute with the same name; for example, a host may have more than one name or more than one network address.

The standard attribute names defined now are as follows. Note that more are likely to be added in the future.

ERROR	The value is an error message. The only error one might expect to get is "no such host".
NAME	The value is a name of the host. There may be more than one NAME attribute; the first one is always the official name, and any additional names are nicknames.
MACHINE-TYPE	The value is the type of machine, such as LISPM, PDP10, etc.
SYSTEM-TYPE	The value is the type of software running on the machine, such as LISPM, ITS, etc.
ARPA	The value is an address of the host on the Arpanet, in the form *host/imp*. The two numbers are decimal.
CHAOS	The value is an address of the host on Chaosnet, as an octal number.
DIAL	The value is an address of the host on Dialnet, as a telephone number.
LCS	The value is an address of the host on the LCSnet, as two octal numbers separated by a slash.
SU	The value is an address of the host on the SUnet, in the form *net#host*. The two numbers are octal.

26.9.12 Dover

A press file may be sent to the Dover printer at MIT by connecting to contact name DOVER at host AI-CHAOS-11. This host provides a protocol translation service that translates from Chaosnet stream protocol to the EFTP protocol spoken by the Dover printer. Only one file at a time can be sent to the Dover, so an attempt to use this service may be refused by a CLS packet containing the string "BUSY". Once the connection has been established, the press file is transmitted as a sequence of 8-bit bytes in data packets (opcode 200). It is necessary to provide packets rapidly enough to keep the Dover's program (Spruce) from timing out; a packet every five seconds suffices. Of course, packets are normally transmitted much more rapidly.

Once the file has been transmitted, an EOF packet must be sent. The transmitter must wait for that EOF to be acknowledged, then send a second one, and then close the connection. The two EOF's are necessary to provide the proper connection-closing sequence for the EFTP protocol. Once the press file has been transmitted to the Dover in this way and stored on the Dover's local disk, it will be processed, prepared for printing, and printed.

If an error message is returned by the Dover while the press file is being transmitted, it is reported back through the Chaosnet as a LOS containing the text of the error message. Such errors are fairly common; the sender of the press file should be prepared to retry the operation a few times.

Most programs that send press files to the Dover first wait for the Dover to be idle, using the Foreign Protocol mechanism of Chaosnet to check the status of the Dover. This is optional, but is courteous to other users since it prevents printing from being held up while additional files are sent to the Dover and queued on its local disk.

It would be possible to send to a press file to the Dover using its EFTP protocol through the Foreign Protocol mechanism, rather than using the AI-CHAOS-11 gateway service. This is not usually done because EFTP, which requires a handshake for every packet, tends to be very slow on a timesharing system.

26.9.13 Remote Disk

The Remote Disk server exists on Lisp Machines to allow other machines to refer to or modify the contents of the Lisp Machine's disk. Primarily this is used for printing and editing the disk label.

After first establishing a connection to contact name `REMOTE-DISK`, the user process sends commands as packets which contain a line of text, ending with a Return character. The text consists of a command name, a space, and arguments interpreted according to the command. The server processing the command may send disk data to the user, or it may read successive packets and write them to the disk. It is up to the user to know how many packets of disk data to read or send after each command. The commands are:

READ *unit block n-blocks*
: Reads *n-blocks* of data from disk *unit* starting at *block* and transmits their contents to the user process.

WRITE *unit block n-blocks*
: Reads data from the net connection and stores it into *n-blocks* disk blocks on disk *unit* starting at *block*.

SAY *text*
: Prints *text*, which is simply all the rest of the line following SAY, on the screen of the server host as a notification.

Each disk block is transmitted as three packets, the first two containing the data for 121 (decimal) Lisp Machine words, and the third containing the data for the remaining 14 (decimal) words of the disk block. Each packet's data ends with a checksum made by adding together all the 8-bit bytes of the actual disk data stored in the packet.

26.9.14 The Eval Server

The Eval server is available on Lisp Machines with contact name `EVAL`. It provides a read-eval-print loop which reads and prints using the Chaosnet connection. The data consists of text in the ASCII character set.

Each time a complete s-expression arrives, the Eval server reads it, evaluates it and prints the list of values back onto the network connection, followed by a CRLF. There is no way for the user process to tell the end of the output for a particular s-expression; the usual application is simply to copy all the output to a user's terminal asynchronously.

The Eval server is disabled when the Lisp Machine is logged in, unless the user requests to enable it.

chaos:eval-server-on *mode*
Turn the Eval server on this Lisp Machine on or off. *mode* can be t (on), nil (off), or :notify (on, but notify the user when a connection is made).

26.10 Using Higher Level Protocols

qsend *user* &optional *text*
Sends a message to another user. qsend is different from mail because it sends the message immediately; it will appear within seconds on the other user's screen, rather than being saved in her mail file.

user should be a string of the form "*username*@*hostname*"; *host* is the name of the Lisp Machine or timesharing system the user is currently logged-in to. Multiple recipients separated by commas are also allowed. *text* is a string which is the message. If *text* is not specified, you are prompted to type in a message.

Unlike mail and bug, qsend does not put up a window to allow you to compose the message; it just reads it from the input stream. Use Converse if you wish to compose sends in the editor. Converse can be invoked by typing System C. If you have started typing in a message to qsend, you can switch to Converse by typing Control-Meta-E ("Edit"). The text you have typed so far is transferred into Converse.

qsend does give you the ability to insert the text of the last message you received. Type Control-Meta-Y to do this.

reply &optional *text*
qreply &optional *text*
Sends *text* as a message to the last user who sent a message to you, like qsend with an appropriate first argument provided. The two names are synonymous.

chaos:shout &optional *message*
Sends *message* to every Lisp Machine at your site. If you do not specify *message*, it is read from *standard-input*.

print-sends
Reprints any messages that have been received. This is useful if you want to see a message again.

supdup &optional *host*
host may be a string or symbol, which is taken as a host name, or a number, which is taken as a host number. If no *host* is given, the machine you are logged-in to is assumed. This function opens a connection to the host over the Chaosnet using the Supdup protocol, and allows the Lisp Machine to be used as a terminal for any ITS, UNIX or TOPS-20 system.

To give commands to **supdup**, type the **Network** key followed by one character. Type **Network** followed by **Help** for documentation.

telnet &optional *host simulate-imlac*
telnet is similar to **supdup** but uses the Arpanet-standard Telnet protocol, simulating a printing terminal rather than a display terminal.

hostat &rest *hosts*
Asks each of the *hosts* for its status using the STATUS protocol, and prints the results. If no hosts are specified, all hosts on the Chaosnet are asked. Hosts can be specified either by name or by number.

For each host, a line is output that either says that the host is not responding or gives metering information for the host's network attachments. If a host is not responding, that usually means that it is down or there is no such host at that address. A Lisp Machine can fail to respond if it is looping inside **without-interrupts** or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

finger &optional *spec* (*stream* ***standard-output***)
whois &optional *spec* (*stream* ***standard-output***)
Prints brief (finger) or verbose (whois) information about a user or users specified by *spec*, on *stream*. *spec* can be a user name, @ followed by a host name, or a user name, @, and a host name. If there is no host name, the default login host is used. If there is no user name, all users on the host are described.
Examples:

```
(finger "@OZ")
(whois "RMS@OZ")
```

chaos:finger-all-lms &optional *stream print-free return-free hosts*
Prints a line of information about the user of each Lisp Machine in *hosts* (the default is all Lisp Machines at this site) on *stream* (default is ***standard-output***).

If *print-free* is non-nil, information on free Lisp Machines and nonresponding Lisp Machines is also printed.

If *return-free* is non-nil, then this function returns two values, the first a list of host objects of free Lisp Machines, the second a list of host objects of nonresponding Lisp Machines.

chaos:user-logged-into-host-p *username host*
Returns **t** if there is a user named *username* logged in on *host* (a host name or host object).

chaos:find-hosts-or-lispms-logged-in-as-user *user hosts*
Return a list of host objects for hosts on which *user* is logged in. All Lisp Machines at this site are checked, and so are *hosts* (which are presumably non-Lisp machines).

tv:close-all-servers *reason*

Close the connections of all network servers on this Lisp Machine, giving *reason* (a string) as the reason in the CLS packet.

Note that PEEK has a mode that displays information on the active network servers.

27. 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.

The Lisp Machine consists of 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 a session must be loaded into the same environment. Each of these programs is composed of a group of functions; apparently 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 which 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 we are to enable two programs to coexist in the Lisp world, each with its own function **pull**, then each program must have its own symbol named **pull**, because there can't be two function definitions on the same symbol. This means that separate *name spaces*—mappings between names and symbols—must be provided for the two programs. The package system is designed to do just that.

Under the package system, the author of a program or a group of closely related programs identifies them together as a *package*. The package system associates a distinct name space with each package.

Here is an example: 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 (or something). 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 feature can be used to provide a separate name space 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, its symbols would be entered in the **chaos** package's name space. So when the Chaosnet program's **get-packet** referred to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** name space would be found, which would be 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**.

In order to have multiple name spaces, the function **intern**, which searches for a name, must allow the name space to be specified. **intern** accepts an optional second argument which is the package to search.

It's obvious that every file has to be loaded into the right package to serve its purpose. It may not be so obvious that every file must be compiled in the right package, but it's just as true. Luckily, this usually happens automatically.

The system can get the package of a source file from its -*- line. For instance, you can put at the front of your file a line such as

```
; -*- Mode:Lisp; Package:System-Internals -*-
```

The compiler puts the package name into the QFASL file for use when it is loaded. If a file doesn't have such a package specification in it, the system loads it into the current package and tells you what it did.

27.1 The Current Package

At any time, one package is the *current package*. By default, symbol lookup happens in the current package.

package *Variable*
package *Variable*

The value of the this variable is the current package. intern searches this package if it is not given a second argument. Many other functions for operating on packages also use this as the default.

Setting or binding the variable changes the current package. May the Goddess help you if you set it to something that isn't a package!

The two names are synonymous.

Each process or stack group can have its own setting for the current package by binding *package* with let. The actual current package at any time is the value bound by the process which is running. The bindings of another process are irrelevant until the process runs.

pkg-bind *pkg body...* *Macro*

pkg may be a package or a package name. The forms of the *body* are evaluated sequentially with the variable *package* bound to the package named by *pkg*.

Example:

```
(pkg-bind "ZWEI"
  (read-from-string function-name))
```

When a file is loaded, *package* is bound to the correct package for the file (the one named in the file's -*- line). The Chaosnet program file has Package: Chaos; in the -*- line, and therefore its symbols are looked up in the chaos package. A QFASL file has an encoded representation of the -*- line of the source file; it looks different, but it serves the same purpose.

The current package is also relevant when you type Lisp expressions on the keyboard; it controls the reading of the symbols that you type. Initially it is the package user. You can select a different package using pkg-goto, or even by setqing *package*. If you are working with the Chaosnet program, it might be useful to type (pkg-goto 'chaos) so that your symbols are found in the chaos package by default. The Lisp listen loop binds *package* so that pkg-goto in one Lisp listener does not affect others, or any other processes whatever.

pkg-goto *package* &optional *globally*

Sets ***package*** to *package*, if *package* is suitable. (Autoexporting packages used by other packages are not suitable because it you could cause great troubles by interning new symbols in them). *package* may be specified as a package object or the name of one. If **globally** is non-nil, then this function also calls **pkg-goto-globally** (see below)

The Zmacs editor records the correct package for each buffer; it is determined from the file's -*- line. This package is used whenever expressions are read from the buffer. So if you edit the definition of the Chaosnet **get-packet** and recompile it, the new definition is read in the **chaos** package. The current buffer's package is also used for all expressions or symbols typed by the user. Thus, if you type **Meta-. allocate-pbuf** while looking at the Chaosnet program, you get the definition of the **allocate-pbuf** function in the **chaos** package.

The variable ***package*** also has a global binding, which is in effect in any process or stack group which does not rebind the variable. New processes that do bind ***package*** generally use the global binding to initialize their own bindings, doing **(let ((*package* *package*)) ...)**. Therefore, it can be useful to set the global binding. But you cannot do this with **setq** or **pkg-goto** from a Lisp listener, or in a file, because that will set the local binding of ***package*** instead. Therefore you must use **setq-globally** (page 35) or **pkg-goto-globally.**

pkg-goto-globally *package*

Sets the global binding of ***package*** to *package*. An error is signaled if *package* is not suitable. Bindings of *package* other than the the global one are not changed, including the current binding if it is not the global one.

The name of the current package is always displayed in the middle of the who line, with a colon following it. This describes the process which the who line in general is describing; normally, the process of the selected window. No matter how the current package is changed, the who line will eventually show it (at one-second intervals). Thus, while a file is being loaded, the who line displays that file's package; in the editor, the who line displays the package of the selected buffer.

27.2 Package Prefixes

The separation of name spaces is not an uncrossable gulf. Consider a program for accessing files, using the Chaosnet. It may be useful to put it in a distinct package **file-access**, not **chaos**, so that the programs are protected from accidental name conflicts. But the file program cannot exist without referring to the functions of the Chaosnet program.

The colon character (':') has a special meaning to the Lisp reader. When the reader sees a colon preceded by the name of a package, it reads the next Lisp object with ***package*** bound to that package. Thus, to refer to the symbol **connect** in package **chaos**, we write **chaos:connect**. Some symbols documented in this manual require package prefixes to refer to them; they are always written with an appropriate prefix.

Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function (for some reason), it could use **arpa:allocate-pbuf**.

Package prefixes are printed on output also. If you would need a package prefix to refer to a symbol on input, then the symbol is printed with a suitable package prefix if it supposed to be printed readably (**prin1**, as opposed to **princ**). Just as the current package affects how a symbol is read, it also affects how the symbol is printed. A symbol available in the current package is never printed with a package prefix.

The printing of package prefixes makes it possible to print list structure containing symbols from many packages and read the text to produce an equal list with the same symbols in it—provided the current package when the text is read is the same one that was current when the text was printed.

The package name in a package prefix is read just like a symbol name. This means that escape characters can be used to include special characters in the package name. Thus, **foo/:bar:test** refers to the symbol **test** in the package whose name is "FOO:BAR", and so does **|FOO:BAR|:test**. Also, letters are converted to upper case unless they are escaped. For this reason, the actual name of a package is normally all upper case, but you can use either case when you write a package prefix.

In Common Lisp programs, simple colon prefixes are supposed to be used only for referring to external symbols (see page 642). To refer to other symbols, one is supposed to use two colons, as in **chaos::lose-it-later**. The Lisp machine tradition is to allow reference to any symbol with a single colon. Since this is upward compatible with what is allowed in Common Lisp, single-colon references are always allowed. However, double-colon prefixes are printed for internal symbols when Common Lisp syntax is in use, so that data printed on a Lisp Machine can be read by other Common Lisp implementations.

27.3 Home Packages of Symbols

Each symbol remembers one package which it belongs to: normally, the first one it was ever interned in. This package is available as **(symbol-package** *symbol***)**.

With **make-symbol** (see page 133) it is possible to create a symbol that has never been interned in any package. It is called an *uninterned symbol*, and it remains one as long as nobody interns it. The package cell of an uninterned symbol contains **nil**. Uninterned symbols print with **#:** as a prefix, as in **#:foo**. This syntax can be used as input to create an uninterned symbol with a specific name; but a new symbol is created each time you type it, since the mechanism which normally makes symbols unique is interning in a package. Thus, **(eq #:foo #:foo)** returns **nil**.

symbol-package *symbol*
  Returns the contents of *symbol*'s package cell, which is the package which owns *symbol*, or **nil** if *symbol* is uninterned.

package-cell-location *symbol*

Returns a locative pointer to *symbol*'s package cell. It is preferable to write

```
(locf (symbol-package symbol))
```

rather than calling this function explicitly.

Printing of package prefixes is based on the contents of the symbol's package cell. If the cell contains the **chaos** package, then **chaos:** is printed as the prefix when a prefix is necessary. As a result of obscure actions involving interning and uninterning in multiple packages, the symbol may not actually be present in **chaos** any more. Then the printed prefix is inaccurate. This cannot be helped. If the symbol is not where it claims to be, there is no easy way to find wherever it might be.

27.4 Keywords

Distinct name spaces are useful for symbols which have function definitions or values, to enable them to be used independently by different programs.

Another way to use a symbol is to check for it with **eq**. Then there is no possibility of name conflict. For example, the function **open**, part of the file system, checks for the symbol **:error** in its input using **eq**. A user function might do the same thing. Then the symbol **:error** is meaningful in two contexts, but these meanings do not affect each other. The fact that a user program contains the code **(eq sym :error)** does not interfere with the function of system code which contains a similar expression.

There is no need to separate name spaces for symbols used in this way. In fact, it would be a disadvantage. If both the Chaosnet program and the Arpanet program wish to recognize a keyword named "address", for similar purposes (naturally), it is very useful for programs that can call either one if it is the *same* keyword for either program. But which should it be? **chaos:address**? **arpa:address**?

To avoid this uncertainty, one package called **keyword** has been set aside for the keywords of all programs. The Chaosnet and Arpanet programs would both look for **keyword:address**, normally written as just **:address**.

Symbols in **keyword** are the normal choice for names of keyword arguments; if you use &key to process them, code is automatically generated to look for for symbols in **keyword**. They are also the normal choice for flavor operation names, and for any set of named options meaningful in a specific context.

keyword and the symbols belonging to it are treated differently from other packages in a couple of ways designed to make them more convenient for this usage.

* Symbols belonging to **keyword** are constants; they always evaluate to themselves. (This is brought about by storing the symbol in its own value cell when the symbol is placed in the package). So you can write just **:error** rather than **':error**. The nature of the application of keywords is such that they would always be quoted if they were not constant.
* A colon by itself is a sufficient package prefix for **keyword**. This is because keywords are the most frequent application of package prefixes.

keywordp *object*
    t if *object* is a symbol which belongs to the keyword package.

There are certain cases when a keyword should *not* be used for a symbol to be checked for with **eq**. Usually this is when the symbol 1) does not need to be known outside of a single program, and 2) is to be placed in shared data bases such as property lists of symbols which may sometimes be in **global** or **keyword**. For example, if the Chaosnet program were to record the existence of a host named CAR by placing an **:address** property on the symbol **:car**, or the symbol **car** (notice that **chaos:car** *is* **car**), it would risk conflicts with other programs that might wish to use the **:address** property of symbols in general. It is better to call the property **chaos:address**.

27.5 Inheritance between Name Spaces

In the simplest (but not the default) case, a package is independent of all other packages. This is not the default because it is not usually useful. Consider the standard Lisp function and variables names, such as **car**: how can the Chaosnet program, using the **chaos** package, access them? One way would be to install all of them in the **chaos** package, and every other package. But it is better to have one table of the standard Lisp symbols and refer to it where necessary. This is called *inheritance*. The single package **global** is the only one which actually contains the standard Lisp symbols; other packages such as **chaos** contain directions to "search **global** too".

Each package has a hash table of the symbols. The symbols in this table are said to be *present* (more explicitly, *present directly*) in the package, or *interned* in it. In addition, each package has a list of other packages to inherit from. By default, this list contains the package **global** and no others; but packages can be added and removed at any time with the functions **use-package** and **unuse-package**. We say that a package *uses* the packages it inherits from. Both the symbols present directly in the package and the symbols it inherits are said to be *available* in the package.

Here's how this works in the above example. When the Chaosnet program is read into the Lisp world, the current package would be the **chaos** package. Thus all of the symbols in the Chaosnet program would be interned in the **chaos** package. If there is a reference to a standard Lisp symbol such as **append**, nothing is found in the **chaos** package's own table; no symbol of that name is present directly in **chaos**. Therefore the packages used by **chaos** are searched, including **global**. Since **global** contains a symbol named **append**, that symbol is found. If, however, there is a reference to a symbol that is not standard, such as **get-packet**, the first time it is used it is not found in either **chaos** or **global**. So **intern** makes a new symbol named **get-packet**, and installs it in the **chaos** package. When **get-packet** is referred to later in the Chaosnet program, **intern** finds **get-packet** immediately in the **chaos** package. **global** does not need to be searched.

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 **global** one: that is, it shares the same one that the Chaosnet program got. However, if it refers to **get-packet**, it does *not* get the same one the Chaosnet program got, because the **chaos** package is presumably not used by **arpa**. The **get-packet** in **chaos** not being available, no symbol is found, so a new one is created and placed in the **arpa** package. Further references in the Arpanet program find that **get-packet**.

This is the desired result: the packages share the standard Lisp symbols only.

Inheritance between other packages can also be useful, but it must be restricted: inheriting only some of the symbols of the used package. If the file access program refers frequently to the advertised symbols of the Chaosnet program—the connection states, such as **open-state**, functions such as **connect**, **listen** and **open-stream**, and others—it might be convenient to be able to refer to these symbols from the **file-access** package without need for package prefixes.

One way to do this is to place the appropriate symbols of the **chaos** package into the **file-access** package as well. Then they can be accessed by the file access program just like its own symbols. Such sharing of symbols between packages never happens from the ordinary operation of packages, but it can be requested explicitly using **import**.

import *symbols* &optional (*package* ***package***)

> Is the standard Common Lisp way to insert a specific symbol or symbols into a package. *symbols* is a symbol or a list of symbols. Each of the specified symbols becomes present directly in *package*.
>
> If a symbol with the same name is already present (directly or by inheritance) in *package*, an error is signaled. On proceeding, you can say whether to leave the old symbol there or replace it with the one specified in **import**.

But importing may not be the best solution. All callers of the Chaosnet program probably want to refer to the same set of symbols: the symbols described in the documentation of the Chaosnet program. It is simplest if the Chaosnet program, rather than each caller, says which symbols they are.

Restricted inheritance allows the **chaos** package to specify which of its symbols should be inheritable. Then **file-access** can use package **chaos** and the desired symbols are available in it.

The inheritable symbols of a package such as **chaos** in this example are called *external*; the other symbols are *internal*. Symbols are internal by default. The function **export** is how symbols are made external. Only the external symbols of a package are inherited by other packages which use it. This is true of **global** as well; Only external symbols in **global** are inherited. Since **global** exists only for inheritance, every symbol in it is external; in fact, any symbol placed in **global** is automatically made external. **global** is said to be *autoexporting*. A few other packages with special uses, such as **keyword** and **fonts**, are autoexporting. Ordinary packages such as **chaos**, which programs are loaded in, should not be.

If a request is made to find a name in a package, first the symbols present directly in that package are searched. If the name is not found that way, then all the packages in the used-list are searched; but only external symbols are accepted. Internal symbols found in the used packages are ignored. If a new symbol needs to be created and put into the name space, it is placed directly in the specified package. New symbols are never put into the inherited packages.

The used packages of a package are not in any particular order. It does not make any difference which one is searched first, because they are *not allowed* to have any conflicts among them. If you attempt to set up an inheritance situation where a conflict would exist, you get an error immediately. You can then specify explicitly how to resolve the conflict. See section 27.7, page 647.

The packages used by the packages used are *not* searched. If package **file-access** uses package **chaos** and file **mypackage** uses package **file-access**, this does not cause **mypackage** to inherit anything from **chaos**. This is desirable; the Chaosnet functions for whose sake **file-access** uses **chaos** are not needed in the programs in **mypackage** simply to enable them to communicate with **file-access**. If it is desirable for **mypackage** to inherit from **chaos**, that can be requested explicitly.

These functions are used to set up and control package inheritance.

use-package *packages* &optional (*in-package* ***package***)

Makes *in-package* inherit symbols from *packages*, which should be either a single package or name for a package, or a list of packages and/or names for packages.

This can cause a name conflict, if any of *packages* has a symbol whose name matches a symbol in *in-package*. In this case, an error is signaled, and you must resolve the conflict or abort.

unuse-package *packages* &optional (*in-package* ***package***)

Makes *in-package* cease to inherit symbols from *packages*.

package-use-list *package*

Returns the list of packages used by *package*.

package-used-by-list *package*

Returns the list of packages which use *package*.

You can add or remove inheritance paths at any time, no matter what else you have done with the package.

These functions are used to make symbols external or internal in a package. By default, they operate on the current package.

export *symbols* &optional (*package* ***package***)

Makes *symbols* external in *package*. *symbols* should be a symbol or string or a list of symbols and/or strings. The specified symbols or strings are interned in *package*, and the symbols found are marked external in *package*.

If one of the specified symbols is found by inheritance from a used package, it is made directly present in *package* and then marked external there. (We know it was already external in the package it was inherited from.)

Note that if a symbol is present directly in several packages, it can be marked external or internal in each package independently. Thus, it is the symbol's presence in a particular package which is external or not, rather than the symbol itself. **export** makes symbols external in whichever package you specify; if the same symbols are present directly in any other package, their status as external or internal in the other package is not affected.

unexport *symbols* &optional (*package* ***package***)
: Makes *symbols* not be external in *package*. An error occurs if any of the symbols fails to be directly present in *package*.

package-external-symbols *package*
: Returns a list of all the external symbols of *package*.

globalize *name-or-symbol* &optional (*into-package* "GLOBAL")

Sometimes it will be discovered that a symbol which ought to be in **global** is not there, and the file defining it has already been loaded, thus mistakenly creating a symbol with that name in some other package. Creating a symbol in **global** would not fix the problem, since pointers to the misbegotten symbol already exist. Even worse, similarly named symbols may have been created mistakenly in other packages by code attempting to refer to the global symbol, and those symbols also are already pointed to. **globalize** is designed for use in correcting such a situation.

globalize *symbol-or-string* &optional (*package* "GLOBAL")
: If *name-or-symbol* is a name (a string), interns the name in *into-package* and then forwards together all symbols with the same name in all the packages that use *into-package* as well as in *into-package* itself. These symbols are forwarded together so that they become effectively one symbol as far as the value, function definition and properties are concerned. The value of the composite is taken from whichever of the symbols had a value; a proceedable error is signaled if multiple, distinct values were found. The function definition is treated similarly, and so is each property that any of the symbols has.

  If *name-or-symbol* is a symbol, **globalize** interns that symbol in *into-package* and then forwards the other symbols to that one.

  The symbol which ultimately is present in *into-package* is also exported.

27.6 Packages and Interning

The most important service of the package system is to look up a name in a package and return the symbol which has that name in the package's name space. This is done by the function **intern**, and is called *interning*. When you type a symbol as input, **read** converts your characters to the actual symbol by calling **intern**.

The function **intern** allows you to specify a package as the second argument. It can be specified by giving either the package object itself or a string or symbol that is a name for the package. **intern** returns three values. The first is the interned symbol. The second is a keyword that says how the symbol was found. The third is the package in which the symbol was actually found. This can be either the specified package or one of its used packages.

When you don't specify the second argument to **intern**, the current package, which is the value of the symbol ***package***, is used. This happens, in particular, when you call **read** and **read** calls **intern**. To specify the package for such functions to use, bind the symbol ***package*** temporarily to the desired package with **pkg-bind**.

There are actually four forms of the intern function: regular **intern**, **intern-soft**, **intern-local**, and **intern-local-soft**. **-soft** means that the symbol should not be added to the package if there isn't already one; in that case, all three values are **nil**. **-local** turns off inheritance; it means that the used packages should not be searched. Thus, **intern-local** can be used to cause shadowing. **intern-local-soft** is right when you want complete control over what packages to search and when to add symbols. All four forms of **intern** return the same three values, except that the **soft** forms return **nil nil nil** when the symbol isn't found.

intern *string-or-symbol* &optional (*pkg* ***package***)

The simplest case of **intern** is where *string-or-symbol* is a string. (It makes a big difference which one you use.) **intern** searches *pkg* and its used packages sequentially, looking for a symbol whose print-name is equal to *string-or-symbol*. If one is found, it is returned. Otherwise, a new symbol with *string-or-symbol* as print name is created, placed in package *pkg*, and returned.

The first value of **intern** is always the symbol found or created. The second value tells whether an existing symbol was found, and how. It is one of these four values:

:internal	A symbol was found present directly in *pkg*, and it was internal in *pkg*.
:external	A symbol was found present directly in *pkg*, and it was external in *pkg*.
:inherited	A symbol was found by inheritance from a package used by *pkg*. You can deduce that the symbol is external in that package.
nil	A new symbol was created

The third value returned by **intern** says which package the symbol found or created is present directly in. This is different from *pkg* if and only if if the second value is **:inherited**.

If *string-or-symbol* is a symbol, the search goes on just the same, using the print-name of *string-or-symbol* as the string to search for. But if no existing symbol is found, *string-or-symbol* itself is placed directly into *pkg*, just as **import** would do. No new symbol is created; *string-or-symbol itself* is the "new" symbol. This is done even if *string-or-symbol* is already present in another package. You can create arbitrary arrangements of sharing of symbols between packages this way.

Note: **intern** is sensitive to case; that is, it will consider two character strings different even if the only difference is one of upper-case versus lower-case. The reason that symbols get converted to upper-case when you type them in is that the reader converts the case of characters in symbols; the characters are converted to upper-case before **intern** is ever called. So if you call **intern** with a lower-case **"foo"** and then with an upper-case **"FOO"**, you won't get the same symbol.

intern-local *string-or-symbol* &optional (*pkg* ***package***)

Like **intern** but ignores inheritance. If a symbol whose name matches *string-or-symbol* is present directly in *pkg*, it is returned; otherwise *string-or-symbol* (if it is a symbol) or a new symbol (if *string-or-symbol* is a string) is placed directly in *pkg*.

intern-local returns second and third values with the same meaning as those of **intern**. However, the second value can never be **:inherited**, and the third value is always *pkg*.

The function **import** is implemented by passing the symbol to be imported to **intern-local**.

intern-soft *string* &optional (*pkg* ***package***)
find-symbol *string* &optional (*pkg* ***package***)

Like **intern** but never creates a symbol or modifies *pkg*. If no existing symbol is found, **nil** is returned for all three values. It makes no important difference if you pass a symbol instead of a string.

intern-soft returns second and third values with the same meaning as those of **intern**. However, if the second value is **nil**, it does not mean that a symbol was created, only that none was found. In this case, the third value is **nil** rather than a package.

find-symbol is the Common Lisp name for this function. The two names are synonymous.

intern-local-soft *string* &optional (*pkg* ***package***)

Like **intern-soft** but without inheritance. If a matching symbol is found directly present in *pkg*, it is returned; otherwise, the value is **nil**.

intern-local-soft returns second and third values with the same meaning as those of **intern**. However, if the second value is **nil**, it does not mean that a symbol was created, only that none was found. Also, it can never be **:inherited**. The third value is rather useless as it is either *pkg*, or **nil** if the second value is **nil**.

remob *symbol* &optional (*package* (**symbol-package** *symbol*))
unintern *symbol* &optional (*package* ***package***)

Both remove *symbol* from *package*. *symbol* itself is unaffected, but **intern** will no longer find it in *package*. *symbol* is not removed from any other package, even packages used by *package*, if it should be present in them. If *symbol* was present in *package* (and therefore, was removed) then the value is **t**; otherwise, the value is **nil**.

In **remob**, *package* defaults to the contents of the symbol's package cell, the package it belongs to. In **unintern**, *package* defaults to the current package. **unintern** is the Common Lisp version and **remob** is the traditional version.

If *package* is the package that *symbol* belongs to, then *symbol* is marked as uninterned: **nil** is stored in its package cell.

If a shadowing symbol is removed, a previously-hidden name conflict between distinct symbols with the same name in two used packages can suddenly be exposed, like a discovered check in chess. If this happens, an error is signaled.

27.7 Shadowing and Name Conflicts

In a package that uses **global**, it may be desirable to avoid inheriting a few standard Lisp symbols. Perhaps the user has defined a function **copy-list**, knowing that this symbol was not in **global**, and then a system function **copy-list** was created as part of supporting Common Lisp. Rather than changing the name in his program, he can *shadow* **copy-list** in the program's package. Shadowing a symbol in a package means putting a symbol in that package which hides any symbols with the same name which could otherwise have been inherited there. The symbol is explicitly marked as a *shadowing symbol* so that the name conflict does not result in an error.

Shadowing of symbols and shadowing of bindings are quite distinct. The same word is used for them because they are both examples of the general abstract concept of shadowing, which is meaningful whenever there is inheritance.

Shadowing can be done in the definition of a package (see page 652) or by calling the function **shadow**. **(shadow "COPY-LIST")** creates a new symbol named **copy-list** in the current package, regardless of any symbols with that name already available through inheritance. Once the new symbol is present directly in the package and marked as a shadowing symbol, the potentially inherited symbols are irrelevant.

shadow *names* &optional (*package* ***package***)

> Makes sure that shadowing symbols with the specified names exist in *package*. *names* is either a string or symbol or a list of such. If symbols are used, only their names matter; they are equivalent to strings. Each name specified is handled independently as follows:
>
> If there is a symbol of that name present directly in *package*, it is marked as a shadowing symbol, to avoid any complaints about name conflicts.
>
> Otherwise, a new symbol of that name is created and interned in *package*, and marked as a shadowing symbol.

Shadowing must be done before programs are loaded into the package, since if the programs are loaded without shadowing first they will contain pointers to the undesired inherited symbol. Merely shadowing the symbol at this point does not alter those pointers; only reloading the program and rebuilding its data structures from scratch can do that.

If it is necessary to refer to a shadowed symbol, it can be done using a package prefix, as in **global:copy-list**.

Shadowing is not only for symbols inherited from **global**; it can be used to reject inheritance of any symbol. Shadowing is the primary means of resolving *name conflicts* in which there multiple symbols with the same name are available, due to inheritance, in one package.

Name conflicts are not permitted to exist unless a resolution for the conflict has been stated in advance by specifying explicitly which symbol is actually to be seen in package. If no resolution has been specified, any command which would create a name conflict signals an error instead.

For example, a name conflict can be created by **use-package** if it adds a new used package with its own symbol **foo** to a package which already has or inherits a different symbol with the same name **foo**. **export** can cause a name conflict if the symbol becoming external is now supposed to be inherited by another package which already has a conflicting symbol. On either occasion, if shadowing has not already been performed to control the outcome, an error is signaled and the useage or exportation does not occur.

The conflict is resolved—in advance, always—by placing the preferred choice of symbol in the package directly, and marking it as a shadowing symbol. This can be done with the function shadowing-import. (Actually, you can proceed from the error and specify a resolution, but this works by shadowing and retrying. From the point of view of the retried operation, the resolution has been done in advance.)

shadowing-import *symbols* &optional (*package* *package*)
Interns the specified symbols in *package* and marks them as shadowing symbols. *symbols* must be a list of symbols or a single symbol; strings are not allowed.

Each symbol specified is placed directly into *package*, after first removing any symbol with the same name already interned in *package*. This is rather drastic, so it is best to use shadowing-import right after creating a package, when it is still empty.

shadowing-import is primarily useful for choosing one of several conflicting external symbols present in packages to be used.

Once a package has a shadowing symbol named foo in it, any other potentially conflicting external symbols with name foo can come and go in the inherited packages with no effect. It is therefore possible to perform the use-package of another package containing another foo, or to export the foo in one of the used packages, without getting an error.

In fact, shadow also marks the symbol it creates as a shadowing symbol. If it did not do so, it would be creating a name conflict and would always get an error.

package-shadowing-symbols *package*
Returns the list of shadowing symbols of *package*. Each of these is a symbol present directly in *package*. When a symbol is present directly in more than one package, it can be a shadowing symbol in one and not in another.

27.8 Styles of Using Packages

The unsophisticated user need never be aware of the existence of packages when writing his programs. His files are loaded into package user by default, and keyboard input is also read in user by default. Since all the functions that unsophisticated users are likely to need are provided in the global package, which user inherits from, they are all available without special effort. In this manual, functions that are not in the global package are documented with colons in their names, and they are all external, so typing the name the way it is documented does work in both traditional and Common Lisp syntax.

However, if you are writing a generally useful tool, you should put it in some package other than user, so that its internal functions will not conflict with names other users use. If your program contains more than a few files, it probably should have its own package just on the chance that someone else will use it someday along with other programs.

If your program is large, you can use multiple packages to help keep its modules independent. Use one package for each module, and export from it those of the module's symbols which are reasonable for other modules to refer to. Each package can use the packages of other modules that it refers to frequently.

27.9 Package Naming

A package has one name, also called the *primary name* for extra clarity, and can have in addition any number of *nicknames*. All of these names are defined globally, and all must be unique. An attempt to define a package with a name or nickname that is already in use is an error.

Either the name of a package or one of its nicknames counts as a *name for* the package. All of the functions described below that accept a package as an argument also accept a name for a package (either as a string, or as a symbol whose print-name is the name). Arguments that are lists of packages may also contain names among the elements.

When the package object is printed, its primary name is used. The name is also used by default when printing package prefixes of symbols. However, when you create the package you can specify that one of the nicknames should be used instead for this purpose. The name to be used for this is called the *prefix name*.

Case is significant in package name lookup. Usually package names should be all upper case. read converts package prefixes to upper case except for quoted characters, just as it does to symbol names, so the package prefix will match the package name no matter what case you type it in, as long as the actual name is upper case: TV:FOO and tv:foo refer to the same symbol. |tv|:foo is different from them, and normally erroneous since there is no package initially whose name is 'tv' in lower case.

In the functions find-package and pkg-find-package, and others which accept package names in place of packages, if you specify the name as a string you must give it in the correct case:

```
    (find-package "TV") => the tv package
    (find-package "tv") => nil
```

You can alternatively specify the name as a symbol; then the symbol's pname is used. Since read converts the symbol's name to upper case, you can type the symbol in either upper or lower case:

```
    (find-package 'TV) => the tv package
    (find-package 'tv) => the tv package
```

since both use the symbol whose pname is "TV".

Relevant functions:

package-name *package*
    Returns the name of *package* (as a string).

package-nicknames *package*

Returns the list of nicknames (strings) of *package*. This does not include the name itself.

package-prefix-print-name *package*

Returns the name to be used for printing package prefixes that refer to *package*.

rename-package *package new-name* &optional *new-nicknames*

Makes *new-name* be the name for *package*, and makes *new-nicknames* (a list of strings, possibly **nil**) be its nicknames. An error is signaled if the new name or any of the new nicknames is already in use for some other package.

find-package *name* &optional *use-local-names-package*

Returns the package which *name* is a name for, or **nil** if there is none. If *use-local-names-package* is non-**nil**, the local nicknames of that package are checked first. Otherwise only actual names and nicknames are accepted. *use-local-names-package* should be supplied only when interpreting package prefixes.

If *name* is a package, it is simply returned.

If a list is supplied as *name*, it is interpreted as a specification of a package name and how to create it. The list should look like

(*name super-or-use size*)

or

(*name options*)

If *name* names a package, it is returned. Otherwise a package is created by passing *name* and the *options* to **make-package**.

pkg-find-package *name* &optional *create-p use-local-names-package*

Invokes **find-package** on *name* and returns the package that finds, if any. Otherwise, a package may be created, depending on *create-p* and possibly on how the user answers. These values of *create-p* are meaningful:

nil	An error is signaled if an existing package is not found.
t	A package is created, and returned.
:find	**nil** is returned.
:ask	The user is asked whether to create a package. If he answers **Yes**, a package is created and returned. If he answers **No**, **nil** is returned.

If a package is created, it is done by calling **make-package** with *name* as the only argument.

This function is not quite for historical compatibility only, since certain values of *create-p* provide useful features.

sys:package-not-found (error) *Condition*

is signaled by pkg-find-package with second argument :error, nil or omitted, when the package does not exist.

The condition instance supports the operations :name and :relative-to; these return whatever was passed as the first and third arguments to pkg-find-package (the package name, and the package whose local nicknames should be searched).

The proceed types that may be available include

:retry	says to search again for the specified name in case it has become defined; if it is still undefined, the error occurs again.
:create-package	says to search again for the specified name, and create a package with that name (and default characteristics) if none exists yet.
:new-name	is accompanied by a name (a string) as an argument. That name is used instead, ignoring any local nicknames. If that name too is not found, another error occurs.
:no-action	(available on errors from within read) says to continue with the entire read as well as is possible without having a valid package.

27.9.1 Local Nicknames for Packages

Suppose you wish to test new versions of the Chaosnet and file access programs. You could create new packages test-chaos and test-file-access, and use them for loading the new versions of the programs. Then the old, installed versions would not be affected; you could still use them to edit and save the files of the new versions. But one problem must be solved: when the new file access program says "chaos:connect" it must get test-chaos:connect rather than the actual chaos:connect.

This is accomplished by making "CHAOS" a local nickname for "TEST-CHAOS" in the context of the package test-file-access. This means that the when a chaos: prefix is encountered while reading in package test-file-access, it refers to test-chaos rather than chaos.

Local nicknames are allowed to conflict with global names and nicknames; in fact, they are rarely useful unless they conflict. The local nickname takes precedence over the global name.

It is necessary to have a way to override local nicknames. If you (pkg-goto 'test-file-access), you may wish to call a function in chaos (to make use of the old, working Chaosnet program). This can be done using #: as the package prefix instead of just :. #: inhibits the use of local nicknames when it is processed. It always refers to the package which is globally the owner of the name that is specified.

#: prefixes are printed whenever the package name printed is also a local nickname in the current package; that is, whenever an ordinary colon prefix would be misunderstood when read back

These are the functions which manage local nicknames.

pkg-add-relative-name *in-pkg name for-pkg*
Defines *name* as a local nickname in *in-pkg* for *for-pkg*. *in-pkg* and *for-pkg* may be packages, symbols or strings.

pkg-delete-relative-name *in-pkg name*
Eliminates *name* as a local nickname in *in-pkg*.

Looking up local nicknames is done with find-package, by providing a non-nil *use-local-names-package* argument.

27.10 Defining Packages

Before any package can be referred to or made current, it must be defined. This is done with the special form defpackage, which tells the package system all sorts of things, including the name of the package, what packages it should use, its estimated size, and some of the symbols which belong in it. The defpackage form is recognized by Zmacs as a definition of the package name.

defpackage *name* &key ... *Macro*
Defines a package named *name*. The alternating keywords and values are passed, unevaluated, to *make-package* to specify the rest of the information about how to construct the package.

If a package named *name* already exists, it is modified insofar as this is possible to correspond to the new definition.

Here are the possible options and their meanings

nicknames	A list of nicknames for the new package. The nicknames should be specified as strings.
size	A number; the new package is initially made large enough to hold at least this many symbols before a rehash is needed.
use	A list of packages or names for packages which the new package should inherit from, or a single name or package. It defaults to just the global package.
prefix-name	Specifies the name to use for printing package prefixes that refer to this package. It must be equal to either the package name or one of the nicknames. The default is to use the name.
invisible	If non-nil, means that this package should not be put on the list *all-packages*. As a result, find-package will not find this package, not by its name and not by any of its nicknames. You can make normal use of the package in all other respects (passing it as the second argument to intern, passing it to use-package to make other packages inherit from it or it from others, and so on).

export
import
shadow
shadowing-import

If any of these arguments is non-nil, it is passed to the function of the same name, to operate on the package. Thus, if *shadow* is ("FOO" "BAR"), then

```
    (shadow this-package '("FOO" "BAR"))
```

is done.

You could accomplish as much by calling **export**, **import**, **shadow** or **shadowing-import** yourself, but it is clearer to specify all such things in one central place, the **defpackage**.

import-from If non-nil, is a list containing a package (or package name) followed by names of symbols to import from that package. Specifying *import-from* as (chaos "CONNECT" "LISTEN") is nearly the same as specifying *import* as (chaos:connect chaos:listen), the difference being that with *import-from* the symbols **connect** and **listen** are not looked up in the **chaos** package until it is time to import them.

super If non-nil, should be a package or name to be the superpackage of the new package. This means that the new package should inherit from that package, and also from all the packages that package inherits from. In addition, the superpackage is marked as autoexporting. Superpackages are obsolete and are implemented for compatibility only.

relative-names An alist specifying the local nicknames to have in this package for other packages. Each element looks like (*localname package*), where *package* is a package or a name for one, and *localname* is the desired local nickname.

relative-names-for-me

An alist specifying local nicknames by which this package can be referred to from other packages. Each element looks like (*package localname*), where *package* is a package name and *localname* is the name to refer to this package by from *package*.

For example, the system package **eh** could have been defined this way:

```
(defpackage "EH" :size 1200
  :use ("GLOBAL" "SYS") :nicknames ("DBG" "DEBUGGER")
  :shadow ("ARG"))
```

It has room initially for at least 1200. symbols, nicknames **dbg** and **debugger**, uses **system** as well as **global**, and contains a symbol named **arg** which is not the same as the **arg** in **global**. You may note that the function **eh:arg** is documented in this manual (see page 734), as is the function **arg** (see page 238).

The packages of our inheritance example (page 642) might have been defined by

```
(defpackage 'chaos :size 1000 :use '(sys global)
   :export ("CONNECT" "OPEN-STREAM" "LISTEN" ...
            "OPEN-STATE" "RFC-RECEIVED-STATE" ...))

(defpackage 'file-access :size 1500
   :use '(chaos global)
   :export ("OPEN-FILE" "CLOSE-FILE" "DELETE-FILE" ...)
   :import (chaos:connect chaos:open-state))

(defpackage 'mypackage :size 400
   :use '(file-access global))
```

It is usually best to put the package definition in a separate file, which should be loaded into the user package. (It cannot be loaded into the package it is defining, and no other package has any reason to be preferred.) Often the files to be loaded into the package belong to one or a few systems; then it is often convenient to put the system definitions in the same file (see chapter 28, page 660).

A package can also be defined by the package attribute in a file's `-*-` line. Normally this specifies which (existing) package to load, compile or edit the file in. But if the attribute value is a list, as in

```
    -*-Package: (foo :size 300 :use (global system)); ...-*-
```

then loading, compiling or editing the file automatically creates package foo, if necessary with the specified options (just like defpackage options). No defpackage is needed. It is wise to use this feature only when the package is used for just a single file. For programs containing multiple files, it is good to make a system for them, and then convenient to put a defpackage near the defsystem.

make-package *name* &key *nicknames size use prefix-name invisible export shadow import shadowing-import import-from super relative-names relative-names-for-me*

Creates and returns new package with name *name*.

The meanings of the keyword arguments are described under defpackage (page 652).

pkg-create-package *name* &optional (*super* *package*) (*size* #o 200)

Creates a new package named *name* of size *size* with superpackage *super*. This function is obsolete.

kill-package *name-or-package*

Kills the package specified or named. It is removed from the list which is searched when package names are looked up.

package-declare *Macro*

package-declare is an older way of defining a package, obsolete but still used.

(package-declare *name superpackage size* nil
option-1 option-2 ...)

creates a package named *name* with initial size *size*.

super specifies the *superpackage* to use for this package. Superpackages were an old way of specifying inheritance; it was transitive, all symbols were inherited, and only one inheritance path could exist. If *super* is **global**, nothing special needs to be done; otherwise, the old superpackage facility is simulated using the *super* argument to **make-package**.

body is now allowed to contain only these types of elements:

(shadow *names***)**
: Passes the names to the function **SHADOW**.

(intern *names***)** Converts each name to a string and interns it in the package.

(refname *refname packagename***)**
: Makes *refname* a local nickname in this package for the package named *packagename*.

(myrefname *packagename refname***)**
: Makes *refname* a local nickname in the package named *packagename* for this package. If *packagename* is **"GLOBAL"**, makes *refname* a global nickname for this package.

(external *names***)**
: Does nothing. This controlled an old feature that no longer exists.

27.11 Operating on All the Symbols in a Package

To find and operate on every symbol present or available in a package, you can choose between iteration macros that resemble **dolist** and mapping functionals that resemble **mapcar**.

Note that all constructs that include inherited symbols in the iteration can process a symbol more than once. This is because a symbol can be directly present in more than one package. If it is directly present in the specified package and in one or more of the used packages, the symbol is processed once each time it is encountered. It is also possible for the iteration to include a symbol that is not actually available in the specified package. If that package shadows symbols present in the packages it uses, the shadowed symbols are processed anyway. If this is a problem, you can explicitly use **intern-soft** to see if the symbol handed to you is really available in the package. This test is not done by default because it is slow and rarely needed.

do-symbols (*var package result-form*) *body...* *Macro*
Executes *body* once for each symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-local-symbols (*var package result-form*) *body...* *Macro*
Executes *body* once for each symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-external-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each external symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-local-external-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each external symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-all-symbols (*var result-form*) *body...* *Macro*

Executes *body* once for each symbol present in any package. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once.

mapatoms *function* &optional (*package* *package*) (*inherited-p* t)

function should be a function of one argument. mapatoms applies *function* to all of the symbols in *package*. If *inherited-p* is non-nil, then the function is applied to all symbols available in *package*, including inherited symbols.

mapatoms-all *function* &optional (*package* "GLOBAL")

function should be a function of one argument. mapatoms-all applies *function* to all of the symbols in *package* and all other packages which use *package*.

It is used by such functions as apropos and who-calls (see page 791)
Example:

```
(mapatoms-all
  #'(lambda (x)
      (and (alphalessp 'z x)
           (print x))))
```

27.12 Packages as Lisp Objects

A package is a conceptual name space; it is also a Lisp object which serves to record the contents of that name space, and is passed to functions such as intern to identify a name space.

packagep *object*

t if object is a package.

all-packages *Variable*

The value is a list of all packages, except for invisible ones (see the *invisble* argument to make-package, page 654).

list-all-packages
A Common Lisp function which returns ***all-packages***.

pkg-global-package *Constant*
pkg-system-package *Constant*
pkg-keyword-package *Constant*
Respectively, the packages named **global**, **system** and **keyword**.

describe-package *package*
Prints everything there is to know about *package*, except for all the symbols interned in it. *package* can be specified as a package or as the name of one.

To see all the symbols interned in a package, do
```
(mapatoms 'print package)
```

27.13 Common Lisp and Packages

Common Lisp does not have **defpackage** or -*- lines in files. One is supposed to use the function **in-package** to specify which package a file is loaded in.

in-package *name* &key *nicknames use*
Creates a package named *name*, with specified nicknames and used packages, or modifies an existing package named *name* to have those nicknames and used packages.

Then ***package*** is set to this package.

Writing a call to **in-package** at the beginning of the file causes ***package*** to be set to that package for the rest of the file.

If you wish to use this technique for the sake of portability, it is best to have a -*- line with a package attribute also. While **in-package** does work for loading and compilation of the file, Zmacs does not respond to it.

In Common Lisp, the first argument to **intern** or **find-symbol** is required to be a symbol.

27.14 Initialization of the Package System

This section describes how the package system is initialized when generating a new software release of the Lisp Machine system; none of this should affect users.

The cold load, which contains the irreduceable minimum of the Lisp system needed for loading the rest, contains the code for packages, but no packages. Before it begins to read from the keyboard, it creates all the standard packages based on information in **si:initial-packages**, applying **make-package** to each element of it. At first all of the packages are empty. The symbols which belong in the packages **global** and **system** are recorded on lists which are made from the files SYS: SYS2; GLOBAL LISP and SYS: SYS2; SYSTEM LISP. Symbols referred to in the cold load which belong in packages other than **si** have strings (package names) in their package slots; scanning through the area which contains all the symbols, the package initializer puts each such symbol into the package it specifies, and all the rest into **si** unless they are already in **global** or **system**.

27.15 Initial Packages

The initially present packages include:

global	Contains advertised global functions.
user	The default current package for the user's type-in.
sys or **system**	Contains internal global symbols used by various system programs. Many system packages use **system**.
si or **system-internals**	Contains subroutines of many advertised system functions. Many files of the Lisp system are loaded in **si**.
compiler	Contains the compiler. **compiler** uses **sys**.
fs or **file-system**	Contains the code that deals with pathnames and accessing files. **fs** uses **sys**.
eh or **dbg**	Contains the error handler and the debugger. Uses **sys**.
cc or **cadr**	Contains the program that is used for debugging another machine. Uses **sys**.
chaos	Contains the Chaosnet controller. Uses **sys**.
tv	Contains the window system. Uses **sys**.
zwei	Contains the editor.
format	Contains the function **format** and its associated subfunctions.
cli	(Common Lisp Incompatible) contains symbols such as **cli:member** which the same pname as symbols in **global** but incompatible definitions.

There are quite a few others, but it would be pointless to list them all.

Packages that are used for special sorts of data:

fonts	Contains the names of all fonts.
format	Contains the keywords for **format**, as well as the code.
keyword	Contains all keyword symbols, symbols always written with a plain colon as a prefix. These symbols are peculiar in that they are automatically given themselves as values.

Here is a picture depicting the initial package inheritance structure

```
                      global                     keyword
                        |
  /------------------------------------\         fonts
  |     |          |           |       |
user  zwei      system      format  (etc)        cli
                   |
           /-----------------------------------\
           |          |     |      |    |      |
    system-internals  eh  chaos  cadr  fs  compiler
```

28. Maintaining Large Systems

When a program gets large, it is often desirable to split it up into several files. One reason for this is to help keep the parts of the program organized, to make things easier to find. It's also useful to have the program broken into small pieces that are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any piece of it changes; if the program is broken up into many files, only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more commands are needed to manipulate it. To load the program, you now have to load several files separately, instead of just loading one file. To compile it, you have to figure out which files need compilation, by seeing which have been edited since they were last compiled, and then you have to compile those files.

What's even more complicated is that files can have interdependencies. You might have a file called **DEFS** that contains some macro definitions (or flavor or structure definitions), and functions in other files might use those macros. This means that in order to compile any of those other files, you must first load the file **DEFS** into the Lisp environment so that the macros will be defined and can be expanded at compile time. You have to remember this whenever you compile any of those files. Furthermore, if **DEFS** has changed, other files of the program may need to be recompiled because the macros may have changed and need to be re-expanded.

This chapter describes the *system* facility, which takes care of all these things for you. The way it works is that you define a set of files to be a *system*, using the **defsystem** special form, described below. This system definition says which files make up the system, which ones depend on the presence of others, and so on. You put this system definition into its own little file, and then all you have to do is load that file and the Lisp environment will know about your system and what files are in it. You can then use the **make-system** function (see page 666) to load in all the files of the system, recompile all the files that need compiling, and so on.

The system facility is very general and extensible. This chapter explains how to use it and how to extend it. This chapter also explains the *patch* facility, which lets you conveniently update a large program with incremental changes.

28.1 Defining a System

defsystem *name* (*keyword args...*)... *Macro*

Defines a system named *name*. The options selected by the keywords are explained in detail later. In general, they fall into two categories: properties of the system and *transformations*. A transformation is an operation such as compiling or loading that takes one or more files and does something to them. The simplest system is a set of files and a transformation to be performed on them.

Here are a few examples.

```
(defsystem mysys
  (:compile-load ("OZ:<GEORGE>PROG1.LISP" "OZ:<GEORGE2>PROG2.LISP")))

(defsystem zmail
  (:name "ZMail")
  (:pathname-default "SYS: ZMAIL;")
  (:package zwei)
  (:module defs "DEFS")
  (:module mult "MULT" :package tv)
  (:module main ("TOP" "COMNDS" "MAIL" "USER" "WINDOW"
                 "FILTER" mult "COMETH"))
  (:compile-load defs)
  (:compile-load main (:fasload defs)))

(defsystem bar
  (:module reader-macros "BAR:BAR;RDMAC")
  (:module other-macros "BAR:BAR;MACROS")
  (:module main-program "BAR:BAR;MAIN")
  (:compile-load reader-macros)
  (:compile-load other-macros (:fasload reader-macros))
  (:compile-load main-program (:fasload reader-macros
                                        other-macros)))
```

The first example defines a new *system* called mysys, which consists of two files, stored on a Tops-20 host names OZ, both of which are to be compiled and loaded. The second example is somewhat more complicated. What all the options mean is described below, but the primary difference is that there is a file DEFS which must be loaded before the rest of the files (main) can be compiled. Also, the files are stored on logical host SYS and directory ZMAIL.

The last example has two levels of dependency. reader-macros must be compiled and loaded before other-macros can be compiled. Both reader-macros and other-macros must then be loaded before main-program can be compiled. All the source files are stored on host BAR, presumably a logical host defined specifically for this system. It is desirable to use a logical host for the files of a system if there is a chance that people at more than one site will be using it; the logical host allows the identical defsystem to be valid at all sites. See section 24.7.5, page 572 for more on logical hosts and logical pathnames.

Note that The defsystem options other than transformations are:

:name Specifies a "pretty" version of the name for the system, for use in printing.

:short-name
: Specified an abbreviated name used in constructing disk label comments and in patch file names for some file systems.

:component-systems
: Specifies the names of other systems used to make up this system. Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems. The format is (:component-systems *names*...).

:package
: Specifies the package in which transformations are performed. A package specified here overrides one in the -*- line of the file in question.

:pathname-default
: Gives a local default within the definition of the system for strings to be parsed into pathnames. Typically this specifies the directory, when all the files of a system are on the same directory.

:warnings-pathname-default
: Gives a default for the file to use to store compiler warnings in, when **make-system** is used with the **:batch** option.

:patchable
: Makes the system be a patchable system (see section 28.8, page 672). An optional argument specifies the directory to put patch files in. The default is the **:pathname-default** of the system.

:initial-status
: Specifies what the status of the system should be when **make-system** is used to create a new major version. The default is **:experimental**. See section 28.8.5, page 679 for further details.

:not-in-disk-label
: Make a patchable system not appear in the disk label comment. This should probably never be specified for a user system. It is used by patchable systems internal to the main Lisp system, to avoid cluttering up the label.

:default-binary-file-type
: Specifies the file type to use for compiled Lisp files. The value you specify should be a string. If you do not specify this, the standard file type **:qfasl** is used.

:module
: Allows assigning a name to a set of files within the system. This name can then be used instead of repeating the filenames. The format is (**:module** *name files options*...). *files* is usually a list of filenames (strings). In general, it is a *module-specification*, which can be any of the following:

  a string
  : This is a file name.

  a symbol
  : This is a module name. It stands for all of the files which are in that module of this system.

  an *external module component*
  : This is a list of the form (*system-name module-names*...), to specify modules in another system. It stands for all of the files which are in all of those modules.

  a list of *module components*
  : A module component is any of the above, or the following:

  a list of file names
  : This is used in the case where the names of the input and output files of a transformation are not related according to the standard naming conventions, for

example when a QFASL file has a different name or resides on a different directory than the source file. The file names in the list are used from left to right, thus the first name is the source file. Each file name after the first in the list is defaulted from the previous one in the list.

To avoid syntactic ambiguity, this is allowed as a module component but not as a module specification.

The currently defined options for the :module clause are

:package Overrides any package specified for the whole system for transformations performed on just this module.

In the second defsystem example above, there are three modules. Each of the first two has only one file, and the third one (main) is made up both of files and another module. To take examples of the other possibilities,

```
(:module prog (("SYS: GEORGE; PROG" "SYS: GEORG2; PROG")))
(:module foo (defs (zmail defs)))
```

The prog module consists of one file, but it lives in two directories, GEORGE and GEORG2. If this were a Lisp program, that would mean that the file SYS: GEORGE; PROG LISP would be compiled into SYS: GEORG2; PROG QFASL. The foo module consists of two other modules the defs module in the same system, and the defs module in the zmail system. It is not generally useful to compile files that belong to other systems; thus this foo module would not normally be the subject of a transformation. However, *dependencies* (defined below) use modules and need to be able to refer to (depend on) modules of other systems.

si:set-system-source-file *system-name filename*

This function specifies which file contains the defsystem for the system *system-name*. *filename* can be a pathname object or a string.

Sometimes it is useful to say where the definition of a system can be found without taking time to load that file. If make-system, or require (page 672), is ever used on that system, the file whose name has been specified will be loaded automatically.

28.2 Transformations

Transformations are of two types, simple and complex. A simple transformation is a single operation on a file, such as compiling it or loading it. A complex transformation takes the output from one transformation and performs another transformation on it, such as loading the results of compilation.

The general format of a simple transformation is (*name input dependencies condition*). *input* is usually a module specification or another transformation whose output is used. The transformation *name* is to be performed on all the files in the module, or all the output files of the other transformation.

dependencies and *condition* are optional.

dependencies is a *transformation specification*, either a list (*transformation-name module-names...*) or a list of such lists. A *module-name* is either a symbol that is the name of a module in the current system, or a list (*system-name module-names...*). A dependency declares that all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. Thus in the zmail example above, the **defs** module must have the **:fasload** transformation performed on it before the **:compile** transformation can be performed on **main**.

The dependency has to be a tranformation that is explicitly specified as a transformation in the system definition, not just an action that might be performed by anything. That is, if you have a dependency (**:fasload foo**), it means that (**fasload foo**) is a tranformation of your system and you depend on that tranformation; it does not simply mean that you depend on **foo**'s being loaded. Furthermore, it doesn't work if (**:fasload foo**) is an implicit piece of another tranformation. For example, the following works:

```
    (defsystem foo
      (:module foo "FOO")
      (:module bar "BAR")
      (:compile-load (foo bar)))
```

but this doesn't work:

```
    (defsystem foo
      (:module foo "FOO")
      (:module bar "BAR")
      (:module blort "BLORT")
      (:compile-load (foo bar))
      (:compile-load blort (:fasload foo)))
```

because **foo**'s **:fasload** is not mentioned explicitly (i.e. at top level) but is only implicit in the (**:compile-load (foo bar)**). One must instead write:

```
    (defsystem foo
      (:module foo "FOO")
      (:module bar "BAR")
      (:module blort "BLORT")
      (:compile-load foo)
      (:compile-load bar)
      (:compile-load blort (:fasload foo)))
```

condition is a predicate which specifies when the transformation should take place. Generally it defaults according to the type of the transformation. Conditions are discussed further on page 671.

The defined simple transformations are:

:fasload Calls the **fasload** function to load the indicated files, which must be QFASL files whose pathnames have canonical type **:qfasl** (see section 24.2.3, page 551). The *condition* defaults to **si:file-newer-than-installed-p**, which is **t** if a newer version of the file exists on the file computer than was read into the current environment.

:readfile Calls the **readfile** function to read in the indicated files, whose names must have canonical type :lisp. Use this for files that are not to be compiled. *condition* defaults to **si:file-newer-than-installed-p**.

:compile Calls the **compile-file** function to compile the indicated files, whose names must have canonical type :lisp. *condition* defaults to **si:file-newer-than-file-p**, which returns t if the source file has been written more recently than the binary file.

A special simple transformation is

:do-components
(:do-components *dependencies*) inside a system with component systems causes the *dependencies* to be done before anything in the component systems. This is useful when you have a module of macro files used by all of the component systems.

The defined complex transformations are

:compile-load (:compile-load *input compile-dependencies load-dependencies compile-condition load-condition*) is the same as (:fasload (:compile *input compile-dependencies compile-condition*) *load-dependencies load-condition*). This is the most commonly-used transformation. Everything after *input* is optional.

:compile-load-init
See page 671.

As was explained above, each filename in an input specification can in fact be a list of strings when the source file of a program differs from the binary file in more than just the file type. In fact, every filename is treated as if it were an infinite list of filenames with the last filename, or in the case of a single string the only filename, repeated forever at the end. Each simple transformation takes some number of input filename arguments and some number of output filename arguments. As transformations are performed, these arguments are taken from the front of the filename list. The input arguments are actually removed and the output arguments left as input arguments to the next higher transformation. To make this clearer, consider the **prog** module above having the :compile-load transformation performed on it. This means that **prog** is given as the input to the :compile transformation and the output from this transformation is given as the input to the :fasload transformation. The :compile transformation takes one input filename argument, the name of a Lisp source file, and one output filename argument, the name of the QFASL file. The :fasload transformation takes one input filename argument, the name of a QFASL file, and no output filename arguments. So, for the first and only file in the **prog** module, the filename argument list looks like ("SYS: GEORGE; PROG" "SYS: GEORG2; PROG" "SYS: GEORG2; PROG" ...). The :compile transformation is given arguments of "SYS: GEORGE; PROG" and "SYS: GEORG2; PROG" and the filename argument list which it outputs as the input to the :fasload transformation is ("SYS: GEORG2; PROG" "SYS: GEORG2; PROG" ...). The :fasload transformation then is given its one argument of "SYS: GEORG2; PROG".

Note that dependencies are not transitive or inherited. For example, if module **a** depends on macros defined in module **b**, and therefore needs **b** to be loaded in order to compile, and **b** has a similar dependency on **c**, **c** need not be loaded for compilation of **a**. Transformations with these dependencies would be written

```
(:compile-load a (:fasload b))
(:compile-load b (:fasload c))
```

To say that compilation of a depends on both b and c, you would instead write

```
(:compile-load a (:fasload b c))
(:compile-load b (:fasload c))
```

If in addition a depended on c (but not b) during loading (perhaps a contains defvars whose initial values depend on functions or special variables defined in c) you would write the transformations

```
(:compile-load a (:fasload b c) (:fasload c))
(:compile-load b (:fasload c))
```

28.3 Making a System

make-system *name* &rest *keywords*

The make-system function does the actual work of compiling and loading. In the example above, if PROG1 and PROG2 have both been compiled recently, then

```
(make-system 'mysys)
```

loads them as necessary. If either one might also need to be compiled, then

```
(make-system 'mysys :compile)
```

does that first as necessary.

The very first thing make-system does is check whether the file which contains the defsystem for the specified system has changed since it was loaded. If so, it offers to load the latest version, so that the remainder of the make-system can be done using the latest system definition. (This only happens if the filetype of that file is LISP.) After loading this file or not, make-system goes on to process the files that compose the system.

If the system name is not recognized, make-system attempts to load the file SYS: SITE; *system-name* SYSTEM, in the hope that that contains a system definition or a call to si:set-system-source-file.

make-system lists what transformations it is going to perform on what files, then asks the user for confirmation. If the user types S when confirmation is requested, then make-system asks about each file individually so that the user can decide selectively which transformations should be performed; then collective reconfirmation is requested. This is like what happens if the :selective keyword is specified. If the user types Y, the transformations are performed. Before each transformation a message is printed listing the transformation being performed, the file it is being done to, and the package. This behavior can be altered by *keywords*.

If the system being made is patchable, and if loading has not been inhibited, then the system's patches are loaded afterward. Loading of patches is silent if the make-system is, and requires confirmation if the make-system does.

These are the keywords recognized by the make-system function and what they do.

:noconfirm	Assumes a yes answer for all questions that would otherwise be asked of the user.

:selective	Asks the user whether or not to perform each transformation that appears to be needed for each file.
:silent	Avoids printing out each transformation as it is performed.
:reload	Bypasses the specified conditions for performing a transformation. Thus files are compiled even if they haven't changed and loaded even if they aren't newer than the installed version.
:noload	Does not load any files except those required by dependencies. For use in conjunction with the :compile option.
:compile	Compiles files also if need be. The default is to load but not compile.
:recompile	This is equivalent to a combination of :compile and :reload: it specifies compilation of all files, even those whose sources have not changed since last compiled.
:no-increment-patch	When given along with the :compile option, disables the automatic incrementing of the major system version that would otherwise take place. See section 28.8, page 672.
:increment-patch	Increments a patchable system's major version without doing any compilations. See section 28.8, page 672.
:no-reload-system-declaration	Turns off the check for whether the file containing the defsystem has been changed. Then the file is loaded only if it has never been loaded before.
:batch	Allows a large compilation to be done unattended. It acts like :noconfirm with regard to questions, turns off more-processing and fdefine-warnings (see inhibit-fdefine-warnings, page 240), and saves the compiler warnings in an editor buffer and a file (it asks you for the name).
:defaulted-batch	This is like :batch except that it uses the default for the pathname to store warnings in and does not ask the user to type a pathname.
:print-only	Just prints out what transformations would be performed; does not actually do any compiling or loading.
:noop	Is ignored. This is useful mainly for programs that call make-system, so that such programs can include forms like `(make-system 'mysys (if compile-p :compile :noop))`

28.4 Adding New Keywords to make-system

make-system keywords are defined as functions on the si:make-system-keyword property of the keyword. The functions are called with no arguments. Some of the relevant variables they can use are

si:*system-being-made* *Variable*
The internal data structure that represents the system being made.

si:*make-system-forms-to-be-evaled-before* *Variable*
A list of forms that are evaluated before the transformations are performed.

si:*make-system-forms-to-be-evaled-after* *Variable*
A list of forms that are evaluated after the transformations have been performed. Transformations can push entries here too.

si:*make-system-forms-to-be-evaled-finally* *Variable*
A list of forms that are evaluated by an unwind-protect when the body of make-system is exited, whether it is completed or not. Closing the batch warnings file is done here. Unlike the si:*make-system-forms-to-be-evaled-after* forms, these forms are evaluated outside of the "compiler warnings context".

si:*query-type* *Variable*
Controls how questions are asked. Its normal value is :normal. :noconfirm means ask no questions and :selective means asks a question for each individual file transformation.

si:*silent-p* *Variable*
If t, no messages are printed out.

si:*batch-mode-p* *Variable*
If t, :batch was specified.

si:*redo-all* *Variable*
If t, all transformations are performed, regardless of the condition functions.

si:*top-level-transformations* *Variable*
A list of the types of transformations that should be performed, such as (:fasload :readfile). The contents of this list are controlled by the keywords given to make-system. This list then controls which transformations are actually performed.

si:*file-transformation-function* *Variable*
The actual function that gets called with the list of transformations that need to be performed. The default is si:do-file-transformations.

si:define-make-system-special-variable *variable value [defvar-p]* *Macro*
Causes *variable* to be bound to *value* during the body of the call to make-system. This allows you to define new variables similar to those listed above. *value* is evaluated on entry to make-system. If *defvar-p* is specified as (or defaulted to) t, *variable* is defined with defvar. It is not given an initial value. If *defvar-p* is specified as *nil*, *variable* belongs to some other program and is not defvar'ed here.

The following simple example adds a new keyword to make-system called :just-warn, which means that fdefine warnings (see page 239) regarding functions being overwritten should be printed out, but the user should not be queried.

```
(si:define-make-system-special-variable
   inhibit-fdefine-warnings inhibit-fdefine-warnings nil)

(defun (:just-warn si:make-system-keyword) ()
  (setq inhibit-fdefine-warnings :just-warn))
```

(See the description of the inhibit-fdefine-warnings variable, on page 240.)

make-system keywords can do something directly when called, or they can have their effect by pushing a form to be evaluated onto si:*make-system-forms-to-be-evaled-after* or one of the other two similar lists. In general, the only useful thing to do is to set some special variable defined by si:define-make-system-special-variable. In addition to the ones mentioned above, user-defined transformations may have their behavior controlled by new special variables, which can be set by new keywords. If you want to get at the list of transformations to be performed, for example, the right way is to set si:*file-transformation-function* to a new function, which then can call si:do-file-transformations with a possibly modified list. That is how the :print-only keyword works.

28.5 Adding New Options for defsystem

Options to defsystem are defined as macros on the si:defsystem-macro property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return nil. There are several variables they can use; the only one of general interest is

si:*system-being-defined* *Variable*

The internal data structure that represents the system that is currently being constructed.

si:define-defsystem-special-variable *variable value* *Macro*

Causes *value* to be evaluated and *variable* to be bound to the result during the expansion of the defsystem special form. This allows you to define new variables similar to the one listed above.

si:define-simple-transformation *Macro*

This is the most convenient way to define a new simple transformation. The form is

(si:define-simple-transformation *name function*
  default-condition input-file-types output-file-types
  pretty-names compile-like load-like)

For example,

```
(si:define-simple-transformation :compile si:qc-file-1
          si:file-newer-than-file-p (:lisp) (:qfasl))
```

input-file-types and *output-file-types* are how a transformation specifies how many input filenames and output filenames it should receive as arguments, in this case one of each. They also, obviously, specify the default file type for these pathnames. The si:qc-file-1 function is mostly like compile-file, except for its interface to packages. It takes input-file and output-file arguments.

pretty-names, *compile-like*, and *load-like* are optional.

pretty-names specifies how messages printed for the user should print the name of the transformation. It can be a list of the imperative ("Compile"), the present participle ("Compiling"), and the past participle ("compiled"). Note that the past participle is not capitalized, because when used it does not come at the beginning of a sentence. *pretty-names* can be just a string, which is taken to be the imperative, and the system will conjugate the participles itself. If *pretty-names* is omitted or nil it defaults to the name of the transformation.

compile-like and *load-like* say when the transformation should be performed. Compile-like transformations are performed when the :compile keyword is given to make-system. Load-like transformations are performed unless the :noload keyword is given to make-system. By default *compile-like* is t but *load-like* is nil.

Complex transformations are defined as normal macro expansions, for example,

```
(defmacro (:compile-load si:defsystem-macro)
                  (input &optional com-dep load-dep
                                   com-cond load-cond)
  '(:fasload (:compile ,input ,com-dep ,com-cond)
             ,load-dep ,load-cond))
```

28.6 More Esoteric Transformations

It is sometimes useful to specify a transformation upon which something else can depend, but which is performed not by default, but rather only when requested because of that dependency. The transformation nevertheless occupies a specific place in the hierarchy. The :skip defsystem macro allows specifying a transformation of this type. For example, suppose there is a special compiler for the read table which is not ordinarily loaded into the system. The compiled version should still be kept up to date, and it needs to be loaded if ever the read table needs to be recompiled.

```
(defsystem reader
  (:pathname-default "SYS: IO;")
  (:package system-internals)
  (:module defs "RDDEFS")
  (:module reader "READ")
  (:module read-table-compiler "RTC")
  (:module read-table "RDTBL")
  (:compile-load defs)
  (:compile-load reader (:fasload defs))
  (:skip :fasload (:compile read-table-compiler))
  (:rtc-compile-load read-table (:fasload read-table-compiler)))
```

Assume that there is a complex transformation :rtc-compile-load, which is like :compile-load except that is is built on a transformation called something like :rtc-compile, which uses the read table compiler rather than the Lisp compiler. In the above system, then, if the :rtc-compile transformation is to be performed, the :fasload transformation must be done on read-table-compiler first, that is the read table compiler must be loaded if the read table is to be recompiled. If you say (make-system 'reader :compile), then the :compile transformation is done on the **read-table-compiler** module despite the **:skip**, compiling the read table compiler if need be. If you say **(make-system 'reader)**, the reader and the read table are loaded, but the **:skip** keeps this from happening to the read table compiler.

So far nothing has been said about what can be given as a *condition* for a transformation except for the default functions, which check for conditions such as a source file being newer than the binary. In general, any function that takes the same arguments as the transformation function (e.g. **compile-file**) and returns **t** if the transformation needs to be performed, can be in this place as a symbol, including for example a closure. To take an example, suppose there is a file that contains **compile-flavor-methods** for a system and that should therefore be recompiled if any of the flavor method definitions change. In this case, the condition function for compiling that file should return **t** if either the source of that file itself or any of the files that define the flavors have changed. This is what the **:compile-load-init** complex transformation is for. It is defined like this:

```
(defmacro (:compile-load-init si:defsystem-macro)
                (input add-dep &optional com-dep load-dep
                 &aux function)
  (setq function (let-closed ((*additional-dependent-modules*
                                add-dep))
                   'compile-load-init-condition))
  '(:fasload (:compile ,input ,com-dep ,function) ,load-dep))

(defun compile-load-init-condition (source-file qfasl-file)
  (or (si:file-newer-than-file-p source-file qfasl-file)
      (local-declare ((special *additional-dependent-modules*))
        (si:other-files-newer-than-file-p
                      *additional-dependent-modules*
                      qfasl-file))))
```

The condition function generated when this macro is used returns **t** either if **si:file-newer-than-file-p** would with those arguments, or if any of the other files in **add-dep**, which presumably is a *module specification*, are newer than the QFASL file. Thus the file (or module) to which the **:compile-load-init** transformation applies will be compiled if it or any of the source files it depends on has been changed, and will be loaded under the normal conditions. In most (but not all cases), **com-dep** is a **:fasload** transformation of the same files as **add-dep** specifies, so that all the files this one depends on will be loaded before compiling it.

28.7 Common Lisp Modules

In Common Lisp, a *module* is a name given to a group of files of code. Modules are not like systems because nothing records what the "contents" of any particular module may be. Instead, one of the files which defines the module contains a **provide** form which says, when that file is loaded, "Module **foo** is now present." Other files may say, using **require**, "I want to use module **foo**."

Normally the **require** form also specifies the files to load if **foo** has not been **provide**'d already. This is where the information of which files are in a module is stored. If the **require** does not have file names in it, the module name **foo** is used in an implementation-dependent manner to find files to load. The Lisp Machine does this by using it as a system name in **make-system**.

provide *module-name*
: Adds *module-name* to the list ***modules*** of modules already loaded. *module-name* should be a string; case is significant.

require *module-name* &rest *files*
: If module *module-name* is not already loaded (on ***modules***), *files* are loaded in order to make the module available. *module-name* should be a string; case is significant. The elements of *files* should be pathnames or namestrings. If *files* is **nil**, **(make-system** *module-name* **:noconfirm)** is done. Note, however, that case is not significant in the argument to **make-system**.

modules *Variable*
: A list of names (strings) of all modules **provide**'d so far.

28.8 The Patch Facility

The patch facility allows a system maintainer to manage new releases of a large system and issue patches to correct bugs. It is designed to be used to maintain both the Lisp Machine system itself and applications systems that are large enough to be loaded up and saved on a disk partition.

When a system of programs is very large, it needs to be maintained. Often problems are found and need to be fixed, or other little changes need to be made. However, it takes a long time to load up all of the files that make up such a system, and so rather than having every user load up all the files every time he wants to use the system, usually the files just get loaded once into a Lisp world, which is then saved away on a disk partition. Users then use this disk partition, copies of which may appear on many machines. The problem is that since the users don't load up the system every time they want to use it, they don't get all the latest changes.

The purpose of the patch system is to solve this problem. A *patch* file is a little file that, when you load it, updates the old version of the system into the new version of the system. Most often, patch files just contain new function definitions; old functions are redefined to do their new thing. When you want to use a system, you first use the Lisp environment saved on the disk, and then you load all the latest patches. Patch files are very small, so loading them doesn't take much time. You can even load the saved environment, load up the latest patches, and then save it away, to save future users the trouble of even loading the patches. (Of course, new patches may be made later, and then these will have to be loaded if you want to get the very latest version.)

For every system, there is a series of patches that have been made to that system. To get the latest version of the system, you load each patch file in the series, in order. Sooner or later, the maintainer of a system wants to stop building more and more patches, and recompile everything, starting afresh. A complete recompilation is also necessary when a system is changed in a far-reaching way, that can't be done with a small patch; for example, if you completely reorganize a program, or change a lot of names or conventions, you might need to completely recompile it to make it work again. After a complete recompilation has been done, the old patch files are no longer suitable to use; loading them in might even break things.

The way all this is kept track of is by labelling each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made; the patch is identified by the major and minor version number together. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number. Thus, patch 93.9 is for major version 93 and minor version 9; it is followed by patch 93.10.

To clarify this, here is a typical scenario. A new system is created; its initial version number is 1.0. Then a patch file is created; the version of the program that results from loading the first patch file into version 1.0 is called 1.1. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2. Then the entire system is recompiled, creating version 2.0 from scratch. Now the two patch files are irrelevant, because they fix old software; the changes that they reflect are integrated into system 2.0.

Note that the second patch file should only be loaded into system 1.1 in order to create system 1.2; you shouldn't load it into 1.0 or any other system besides 1.1. It is important that all the patch files be loaded in the proper order, for two reasons. First, it is very useful that any system numbered 1.1 be exactly the same software as any other system numbered 1.1, so that if somebody reports a bug in version 1.1, it is clear just which software is being complained about. Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

The patch facility keeps track of all the patch files that exist, remembering which version each one creates. There is a separate numbered sequence of patch files for each major version of each system. All of them are stored in the file system, and the patch facility keeps track of where they all are. In addition to the patch files themselves, there are *patch directory* files that contain the patch facility's data base by which it keeps track of what minor versions exist for a major version, and what the last major version of a system is. These files and how to make them are described below.

In order to use the patch facility, you must define your system with **defsystem** (see chapter 28, page 660) and declare it as patchable with the **:patchable** option. When you load your system (with **make-system**, see page 666), it is added to the list of all systems present in the world. The patch facility keeps track of which version of each patchable system is present and where the data about that system reside in the file system. This information can be used to update the Lisp world automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to be loaded, ask which patches are already loaded, and add new patches.

You can also load in patches or whole new systems and then save the entire Lisp environment away in a disk partition. This is explained on section 35.11, page 804.

When a Lisp Machine is booted, it prints out a line of information for each patchable system present in the booted Lisp world, saying which major and minor versions are loaded. This is done by print-herald (see page 674).

print-system-modifications &rest *system-names*

With no arguments, this lists all the systems present in this world and, for each system, all the patches that have been loaded into this world. For each patch it shows the major version number (which is always the same since a world can only contain one major version), the minor version number, and an explanation of what the patch does, as typed in by the person who made the patch.

If print-system-modifications is called with arguments, only the modifications to the systems named are listed.

print-herald &optional *format-dest*

Prints the names and loaded version numbers of all patchable systems loaded, and the microcode. Also printed are the number of the band you booted, the amount of physical and virtual memory you have, the host name of the machine, and its associated machine name. Example:

```
MIT System, band 7 of CADR-1.
640K physical memory, 16127K virtual memory.
 System        98.43
 CADR          3.6
 ZMail         53.10
 MIT-Specific  22.0
 Microcode     309
MIT Lisp Machine One, with associated machine OZ.
```

format-dest defaults to t; if it is nil the answer is returned as a string rather than printed out. *format-dest* can also be a stream to print on.

si:get-system-version &optional *system*

Returns two values, the major and minor version numbers of the version of *system* currently loaded into the machine, or nil if that system is not present. *system* defaults to "System".

si:system-version-info &optional (*brief-p* nil)

Returns a string giving information about which systems and what versions of the systems are loaded into the machine, and what microcode version is running. A typical string for it to produce is:

```
"System 98.48, CADR 3.6, MIT-Specific 22.0, microcode 309"
```

If *brief-p* is t, it uses short names, suppresses the microcode version, any systems which should not appear in the disk label comment, the name System, and the commas:

```
"98.48"
```

28.8.1 Defining a System

In order to use the patch facility, you must declare your system as patchable by giving the :patchable option to defsystem (see chapter 28, page 660). The major version of your system in the file system is incremented whenever make-system is used to compile it. Thus a major version is associated with a set of QFASL files. The major version of your system that is remembered as having been loaded into the Lisp environment is set to the major version in the file system whenever make-system is used to load your system and the major version in the file system is greater than what you had loaded before.

After loading your system, you can save it with the disk-save function (see page 807). disk-save asks you for any additional information you want printed as part of the greeting when the machine is booted. This is in addition to the names and versions of all the systems present in this world. If the system version does not fit in the partition comment field allocated in the disk label, disk-save asks you to type in an abbreviated form.

28.8.2 Loading Patches

load-patches &rest *options*

This function is used to bring the current world up to the latest minor version of whichever major version it is, for all systems present, or for certain specified systems. If there are any patches available, load-patches offers to read them in. With no arguments, load-patches updates all the systems present in this world. If you do not specify the systems to operate on, load-patches also reloads the site files if they have changed (section 35.12, page 810), and reloads the files defining logical host translations if they have changed (page 574).

options is a list of keywords. Some keywords are followed by an argument. The following options are accepted:

:systems *list*	*list* is a list of names of systems to be brought up to date. If this option is not specified, all patchable systems loaded are processed.
:unreleased	Loads unreleased patches with no special querying. These patches should be loaded for experimental use if you wish the benefit of the latest bug fixes, but should not be loaded if you plan to save a band.
:site	Loads the latest site files if they have been changed since last loaded. This is the default if you do not specify explicitly which systems to process.
:nosite	Prevents loading of site files. This is the default when you specify the systems to process.
:hosts	Reloads the files defining logical host translations if they have been changed since last loaded. This is the default if you do not specify explicitly which systems to process.
:nohosts	Prevents loading of logical host translation files. This is the default when you specify the systems to process.

:verbose	Prints an explanation of what is being done. This is the default.
:selective	For each patch, says what it is and then ask the user whether or not to load it. This is the default. If the user answers P (for 'Proceed'), selective mode is turned off for any remaining patches to the current system.
:noselective	Turns off :selective.
:silent	Turns off both :selective and :verbose. In :silent mode all necessary patches are loaded without printing anything and without querying the user.
:force-unfinished	Loads patches that have not been finished yet, if they have been compiled.

load-patches returns t if any patches were loaded.

When you load a patchable system with make-system, load-patches is called automatically on that system.

si:patch-loaded-p *major-version minor-version* &optional (*system-name* "SYSTEM")
Returns t if the changes in patch *major-version.minor-version* of system *system-name* are loaded. If *major-version* is the major version of that system which is currently loaded, then the changes in that patch are loaded if the current minor version is greater than or equal to *minor-version*. If the currently loaded major version is greater than *major-version* then it is assumed that the newer system version contains all the improvements patched into earlier versions, so the value is t.

28.8.3 Making Patches

There are two editor commands that are used to create patch files. During a typical maintenance session on a system you will make several edits to its source files. The patch system can be used to copy these edits into a patch file so that they can be automatically incorporated into the system to create a new minor version. Edits in a patch file can be modified function definitions, new functions, modified defvar's and defconst's, or arbitrary forms to be evaluated, even including load's of new files.

The first step in making a patch is to *start* it. At this stage you must specify which patchable system you are making a patch for. Then you *add* one or more pieces of code from other source files to the patch. Finally you *finish* the patch. This is when you fill in the description of what the patch does; this description is what load-patches prints when it offers to load the patch. If you have any doubts about whether the patch will load and work properly, you finish it *unreleased*; then you can load it to test it but no bands can be saved containing the patch until you explicitly release it later.

It is important that any change you patch should go in a patch for the patchable system to which the changed source file belongs. This makes sure that nobody loads the change into a Lisp world which does not contain the file you were changing—something that might cause trouble.

Also, it ensures that you never patch changes to the same piece of code in two different patchable systems' patches. This would lead to disaster because there is no constraint on the order in which patches to two different systems are loaded.

Starting a patch can be done with Meta-X Start Patch. It reads the name of the system to patch with the minibuffer. Meta-X Add Patch can also start a patch, so an explicit Meta-X Start Patch is needed only infrequently.

Meta-X Add Patch adds the region (if there is one) or the current "defun" to the patch file currently being constructed. If you change a function, you should recompile it, test it, then once it works use Add Patch to put it in the patch file. If no patch is being constructed, one is started for you; you must type in the name of the system to patch.

A convenient way to add all your changes to a patch file is to use Meta-X Add Patch Changed Sections or Meta-X Add Patch Buffer Changed Sections. These commands ask you, for each changed function (or each changed function in the current buffer), whether to add it to the patch being constructed. If you use these commands more than once, a function which has been added to the patch and has not been changed since is considered "unchanged".

The patch file being constructed is in an ordinary editor buffer. If you mistakenly Add Patch something that doesn't work, you can select the buffer containing the patch file and delete it. Then later you can Add Patch the corrected version.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. This way if two people are patching a system at the same time, they do not both get the same minor version number.

After testing and patching all of your changes, use Meta-X Finish Patch to install the patch file so that other users can load it. This compiles the patch file if you have not done so yourself (patches are always compiled). It also asks you for a comment describing the reason for the patch; load-patches and print-system-modifications print these comments. If the patch is complex or it has a good chance of causing new problems, you should not use Meta-X Finish Patch; instead, you should make an unreleased patch.

A finished patch can be *released* or *unreleased*. If a patch is unreleased, it can be loaded in the usual manner if the user says 'yes' to a special query, but once it has been loaded the user will be strongly discouraged from saving a band. Therefore, you still have a chance to edit the patch file and recompile it if there is something wrong with it. You can be sure that the old broken patch will not remain permanently in saved bands.

To finish a patch without releasing it, use the command Meta-X Finish Patch Unreleased. Then the patch can be tested by loading it. After a sufficient period for testing, you can release the patch with Meta-X Release Patch. If you discover a bug in the patch after this point, it is not sufficient to correct it in this patch file; you must put the fix in a new patch to correct any bands already saved with the broken version of this patch.

It is a good principle not to add any new features or fix any additional bugs in a patch once that patch is released; change it only to correct problems with that patch. New fixes to other bugs should go in new patches.

You can only be constructing one patch at any time. Meta-X Add Patch automatically adds to the patch you are constructing. But you can start constructing a different patch without finishing the first. If you use the command Meta-X Start Patch while constructing a patch, you are given the option of starting a new patch. The old patch ceases to be the one you are constructing but the patch file remains in its editor buffer. Later, or in another session, you can go back to constructing the first patch with the command Meta-X Resume Patch. This commands asks for both a patchable system name and the patch version to resume constructing. You can simply save the editor buffer of a patch file and resume constructing that patch in a later session. You can even resume constructing a finished patch; though it rarely makes sense to do this unless the patch is unreleased.

If you start to make a patch and change your mind, use the command Meta-X Cancel Patch. This deletes the record that says that this patch is being worked on. It also tells the editor that you are no longer constructing any patch. You can undo a finished (but unreleased) patch by using Resume Patch and then Cancel Patch. If a patch is released, you cannot remove it from saved bands, so it is not reasonable to cancel it at that stage.

28.8.4 Private Patches

A private patch is a file of changes which is not installed to be loaded automatically in sequence by all users. It is loaded only by explicit request (using the function load). A private patch is not associated with any particular patchable system, and has no version number.

To make a private patch, use the editor command Meta-X Start Private Patch. Instead of a patchable system name, you must specify a filename to use for the patch file; since the patch is not to be installed, there is no standard naming convention for it to follow. Add text to the patch using Meta-X Add Patch and finish it using Meta-X Finish Patch. There is no concept of release for private patches so there is no point in using Meta-X Finish Patch Unreleased. There is also no data base recording all private patches, so Meta-X Start Private Patch will resume an existing patch, or even a finished patch. In fact, finishing a private patch is merely a way to write a comment into it and compile it.

Once the private patch file is made, you can load it like any other file.

The private patch facility is just an easy way to copy code from various files into one new file with Patch-File: T in its attribute list (to prevent warnings about redefining functions defined in other files) and compile that file.

28.8.5 System Status

The patch system has the concept of the *status* of a major version of a system. A status keyword is recorded in the Lisp world for each patchable system that is loaded. There is also a current status for each major version of each system, recorded in the patch directory file for that major version. Loading patches updates the status in the Lisp world to match the current status stored in the patch directory. The status in the patch directory is changed with **si:set-system-status**.

The status is displayed when the system version is displayed, in places such as the system greeting message (**print-herald**) and the disk partition comment.

The status is one of the following keywords:

:experimental	The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the **:initial-status** option to **defsystem**.
:released	The system is released for general use. This status produces no extra text in the system greeting and the disk partition comment.
:obsolete	The system is no longer supported.
:broken	This is like **:experimental**, but is used when the system was thought incorrectly to have been debugged, and hence was **:released** for a while.
:inconsistent	Unreleased patches to this system have been loaded. If any patchable system is in this status, **disk-save** demands extra confirmation, and the resulting saved band is identified as "Bad" in its disk partition comment.

si:set-system-status *system status* &optional *major-version*

Changes the current status of a system, as recorded in the patch directory file. *system* is the name of the system. *major-version* is the number of the major version to be changed; if unsupplied it defaults to the version currently loaded into the Lisp world. *status* should be one of the keywords above.

Do not set the current system status to **:inconsistent**. A status of **:inconsistent** is set up in the Lisp world when an unreleased patch is loaded, and once set that way it never changes in that Lisp world. The status recorded in the system's patch directory file should describe the situation where all currently released patches are loaded. It should never be **:inconsistent**.

28.8.6 Patch Files

The patch system maintains several different types of files in the directory associated with your system. This directory is specified to **defsystem** via either the **:patchable** option or the **:pathname-default** option. These files are maintained automatically, but they are described here so that you can know what they are and when they are obsolete and can be deleted.

If the **:patchable** option to **defsystem** had no argument, then the patch data files are stored on the host, device and directory specified as the system's pathname default. The names and types of the filenames are all standard and do not include the name of the system in any way.

If the **:patchable** option to **defsystem** is given an argument, this argument is a file namestring specifying the host, device and directory to use for storing the patch data files. In addition, the system's short name is used in constructing the names of the files. This allows you to store the patch data files for several systems in the same directory.

There are three kinds of files that record patch information:

* the system patch directory

  This file records the current major version number, so that when the system is recompiled a new number can be allocated.

  On Tops-20, this file has, by default, a name like OZ:PS:<MYDIR>PATCH.DIRECTORY, where the host, device, and directory (OZ:PS:<MYDIR>) come from the system's **:pathname-default** as explained above.

  If **:patchable** is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO.PATCH-DIRECTORY, where the host, device and directory come from **:patchable**'s argument.

* the patch directory of a major version

  There is a file of this kind for each major version of the system. It records the patches that have been made for that major version: the minor version, author, description and release status of each one.

  The data in this file are in the form of a printed representation of a Lisp list with two elements. The first is the system status of this major version (**:experimental**, **:released**, **:broken** or **:obsolete**). The second is another list with an element for each patch. The element for a patch is a list of length four: the minor version, the patch description (a string) or nil for an unfinished patch, the author's name (a string), and a flag that is **t** if the patch is unreleased.

  On a Tops-20, for major version 259, this file has, by default, a name like OZ:PS:<MYDIR>PATCH-259.DIRECTORY.

  If **:patchable** is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO-259.PATCH-DIRECTORY.

* the individual patch

  For each patch made, there is a Lisp source file and a QFASL file.

  On a Tops-20, for version 259.12, these files have, by default, names like OZ:PS:<MYDIR>PATCH-259-12.LISP and OZ:PS:<MYDIR>PATCH-259-12.QFASL.

  If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO-259-12.PATCH-DIRECTORY.

On certain types of file systems, slightly different naming conventions are used to keep the names short enough to be legal.

29. Processes

The Lisp Machine supports *multi-processing*; several computations can be executed concurrently by placing each in a separate *process*. A process is like a processor, simulated by software. Each process has its own program counter, its own stack of function calls and its own special-variable binding environment in which to execute its computation. (This is implemented with stack groups; see chapter 13, page 256.)

If all the processes are simply trying to compute, the machine allows them all to run an equal share of the time. This is not a particularly efficient mode of operation since dividing the finite memory and processor power of the machine among several processes certainly cannot increase the available power and in fact wastes some of it in overhead. The typical use for processes is that at any time only one or two are trying to run. The rest are either *waiting* for some event to occur or *stopped* and not allowed to compete for resources.

A process waits for an event by means of the **process-wait** primitive, which is given a predicate function which defines the event being waited for. A module of the system called the process scheduler periodically calls that function. If it returns **nil** the process continues to wait; if it returns **t** the process is made runnable and its call to **process-wait** returns, allowing the computation to proceed.

A process may be *active* or *stopped*. Stopped processes are never allowed to run; they are not considered by the scheduler, and so can never become the current process until they are made active again. The scheduler continually tests the waiting functions of all the active processes, and those which return non-nil values are allowed to run. When you first create a process with **make-process**, it is inactive.

The activity of a process is controlled by two sets of Lisp objects associated with it, called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically keyword symbols and active objects such as windows and other processes are found. A process is considered active when it has at least one run reason and no arrest reasons.

To get a computation to happen in another process, you must first create a process, then say what computation you want to happen in that process. The computation to be executed by a process is specified as an *initial function* and a list of arguments to that function. When the process starts up it applies the function to the arguments. In some cases the initial function is written so that it never returns, while in other cases it performs a certain computation and then returns, which stops the process.

To *reset* a process means to exit its entire computation nonlocally using ***unwind-stack** (see page 82). Some processes are temporary and die when reset. The other, permanent functions start their computations over again when reset. Resetting a process clears its waiting condition, so that if it is active it becomes runnable. To *preset* a function is to set up its initial function (and arguments) and then reset it. This is how you start up a computation in a process.

All processes in a Lisp Machine run in the same virtual address space, sharing the same set of Lisp objects. Unlike other systems that have special restricted mechanisms for inter-process communication, the Lisp Machine allows processes to communicate in arbitrary ways through shared Lisp objects. One process can inform another of an event simply by changing the value of a global variable. Buffers containing messages from one process to another can be implemented as lists or arrays. The usual mechanisms of atomic operations, critical sections, and interlocks are provided (see store-conditional [page 688], without-interrupts [page 684], and process-lock [page 687]).

A process is a Lisp object, an instance of one of several flavors of process (see chapter 21, page 401). The remainder of this chapter describes the operations defined on processes, the functions you can apply to a process, and the functions and variables a program running in a process can use to manipulate its process.

29.1 The Scheduler

At any time there is a set of *active processes*; as described above, these are all the processes that are not stopped. Each active process is either currently running, runnable (ready to run), or waiting for some condition to become true. The active processes are managed by a special stack group called the *scheduler*, which repeatedly examines each active process to determine whether it is waiting or ready to run. The scheduler then selects one process and starts it up.

The process chosen by the scheduler becomes the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable current-process to it. It remains the current process and continues to run until either it decides to wait, or a *sequence break* occurs. In either case, the scheduler stack group is resumed. It then updates the process's run time meters and chooses a new process to run next. This way, each process that is ready to run gets its share of time in which to execute.

Each process has a *priority* which is a number. Most processes have priority zero. Larger numbers give a process more priority. The scheduler only considers the highest priority runnable processes, so if there is one runnable process with priority 20 then no process with lesser priority can run.

The scheduler determines whether a process is runnable by applying the process's *wait-function* to its *wait-argument-list*. If the wait-function returns a non-nil value, then the process is ready to run; otherwise, it is waiting.

A process can wait for some condition to become true by calling process-wait (see page 685). This function sets the process's wait-function and wait-argument-list as specified by the caller, and resumes the scheduler stack group. A process can also wait for just a moment by calling process-allow-schedule (see page 686), which resumes the scheduler stack group but leaves the process runnable; it will run again as soon as all other runnable processes have had a chance.

A sequence break is a kind of interrupt that is generated by the Lisp system for any of a variety of reasons; when it occurs, the scheduler is resumed. The function si:sb-on (see page 687) can be used to control when sequence breaks occur. The default is to sequence break once a second. Thus even if a process never waits and is not stopped, it is forced to return control to the scheduler once a second so that any other runnable processes can get their turn.

The system does not generate a sequence break when a page fault occurs; thus time spent waiting for a page to come in from the disk is "charged" to a process the same as time spent computing, and cannot be used by other processes. It is done this way for the sake of simplicity; this allows the whole implementation of the process system to reside in ordinary virtual memory, so that it does not have to worry specially about paging. The performance penalty is small since Lisp Machines are personal computers, not multiplexed among a large number of processes. Usually only one process at a time is runnable.

A process's wait function is free to touch any data structure it likes and to perform any computation it likes. Of course, wait functions should be kept simple, using only a small amount of time and touching only a small number of pages, or system performance will be impacted since the wait function will consume resources even when its process is not running.

If a wait function gets an error, the error occurs inside the scheduler. If this enters the debugger, all scheduling comes to a halt until the user proceeds or aborts. Aborting in the debugger inside the scheduler "blasts" the current process by giving it a trivial wait function that always returns nil; this prevents recurrence of the same problem. It is best to write wait functions that cannot get errors, by keeping them simple and by arranging for any problems to be detected before the scheduler sees the wait function. process-wait calls the wait function once before giving it to the scheduler, and this often exposes an error before it can interfere with scheduling.

Note well that a process's wait function is executed inside the scheduler stack-group, *not* inside the process. This means that a wait function may not access special variables bound in the process. It is allowed to access global variables. It can access variables bound by a process through the closure mechanism (chapter 12, page 250). If the wait function is defined lexically within the caller of process-wait then it can access local variables through the lexical scoping mechanism. Most commonly any values needed by the wait function are passed to it as arguments.

current-process *Variable*

The value of current-process is the process that is currently executing, or nil while the scheduler is running. When the scheduler calls a process's wait-function, it binds current-process to the process so that the wait-function can access its process.

without-interrupts *body...* *Macro*

The *body* forms are evaluated with inhibit-scheduling-flag bound to t. This is the recommended way to lock out multi-processing over a small critical section of code to prevent timing errors. In other words the body is an *atomic operation*. The values of the last form in the body are ultimately returned.

In this example, list is presumed to be a global variable referred to from two places in the code which different processes will execute.

```
(without-interrupts
  (push item list))

(without-interrupts
  (cond ((memq item list)
         (setq list (delq item list))
         t)
        (t nil)))
```

inhibit-scheduling-flag *Variable*

The value of inhibit-scheduling-flag is normally nil. without-interrupts binds it to t, which prevents process-switching until inhibit-scheduling-flag becomes nil again. It is cleaner to use without-interrupts than to refer directly to this variable.

process-wait *whostate function* &rest *arguments*

This is the primitive for waiting. The current process waits until the application of *function* to *arguments* returns non-nil (at which time process-wait returns). Note that *function* is applied in the environment of the scheduler, not the environment of the process-wait, so special bindings in effect when process-wait was called are *not* be in effect when *function* is applied. Be careful when using any free references in *function*. *whostate* is a string containing a brief description of the reason for waiting. If the who-line at the bottom of the screen is looking at this process, it will show *whostate*.
Example:

```
(process-wait "Buffer"
              #'(lambda (b) (not (zerop (buffer-n-things b))))
              the-buffer)
```

process-sleep *interval*

Waits for *interval* sixtieths of a second, and then returns. It uses process-wait.

sleep *seconds*

Waits *seconds* seconds and then returns. *seconds* need not be an integer. This also uses process-wait.

process-wait-with-timeout *whostate interval function* &rest *arguments*

This is like process-wait except that if *interval* sixtieths of a second go by and the application of *function* to *arguments* is still returning nil, then process-wait-with-timeout returns anyway. The value returned is the value of applying *function* to *arguments*; thus, it is non-nil if the wait condition actually occurred, nil for a time-out.

If *interval* is nil, there is no timeout, and this function is then equivalent to process-wait.

with-timeout (*interval timeout-forms...*) *body...* *Macro*

body is executed with a timeout in effect for *interval* sixtieths of a second. If *body* finishes before that much time elapses, the values of the last form in *body* are returned.

If after *interval* has elapsed *body* has not completed, its execution is terminated with a throw caught by the with-timeout form. Then the *timeout-forms* are evaluated and the values of the last one of them are returned.

For example,

```
(with-timeout ((* 60. 60.) (format *query-io* " ... Yes.") t)
  (y-or-n-p "Really do it? (Yes after one minute) "))
```

is a convenient way to ask a question and assume an answer if the user does not respond promptly. This is a good thing to do for queries likely to occur when the user has walked away from the terminal and expects an operation to finish without his attention.

process-allow-schedule

Resumes the scheduler momentarily; all other processes will get a chance to run before the current process runs again.

sys:scheduler-stack-group *Constant*

This is the stack group in which the scheduler executes.

sys:clock-function-list *Variable*

This is a list of functions to be called by the scheduler 60 times a second. Each function is passed one argument, the number of 60ths of a second since the last time that the functions on this list were called. These functions implement various system overhead operations such as blinking the blinking cursor on the screen. Note that these functions are called inside the scheduler, just as are the functions of simple processes (see page 695). The scheduler calls these functions as often as possible, but never more often than 60 times a second. That is, if there are no processes ready to run, the scheduler calls the clock functions 60 times a second, assuming that, all together, they take less than 1/60 second to run. If there are processes continually ready to run, then the scheduler calls the clock functions as often as it can; usually this is once a second, since usually the scheduler gets control only once a second.

sys:active-processes *Variable*

This is the scheduler's data-structure. It is a list of lists, where the car of each element is an active process or nil and the cdr is information about that process.

sys:all-processes *Variable*

This is a list of all the processes in existence. It is mainly for debugging.

si:initial-process *Constant*

This is the process in which the system starts up when it is booted.

si:sb-on &optional *when*

Controls what events cause a sequence break, i.e. when rescheduling occurs. The following keywords are names of events which can cause a sequence break.

:clock	This event happens periodically based on a clock. The default period is one second. See **sys:%tv-clock-rate**, page 293.
:keyboard	Happens when a character is received from the keyboard.
:chaos	Happens when a packet is received from the Chaosnet, or transmission of a packet to the Chaosnet is completed.

Since the keyboard and Chaosnet are heavily buffered, there is no particular advantage to enabling the :keyboard and :chaos events, unless the :clock event is disabled.

With no argument, si:sb-on returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, nil (which disables sequence breaks entirely since it is the empty list), or a number, which is the internal mask, not documented here.

29.2 Locks

A *lock* is a software construct used for synchronization of two processes. A lock is either held by some process, or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing some other process to seize it. A lock protects some resource or data structure so that only one process at a time can use it.

In the Lisp Machine, a lock is a locative pointer to a cell. If the lock is free, the cell contains nil; otherwise it contains the process that holds the lock. The process-lock and process-unlock functions are written in such a way as to guarantee that two processes can never both think that they hold a certain lock; only one process can ever hold a lock at one time.

process-lock *locative* &optional (*lock-value* current-process) (*whostate* "Lock") *timeout*

This is used to seize the lock that *locative* points to. If necessary, process-lock waits until the lock becomes free. When process-lock returns, the lock has been seized. *lock-value* is the object to store into the cell specified by *locative*, and *whostate* is passed on to process-wait.

If *timeout* is non-nil, it should be a fixnum representing a time interval in 60ths of a second. If it is necessary to wait more than that long, an error with condition name sys:lock-timeout is signaled.

process-unlock *locative* &optional (*lock-value* current-process)

This is used to unlock the lock that *locative* points to. If the lock is free or was locked by some other process, an error is signaled. Otherwise the lock is unlocked. *lock-value* must have the same value as the *lock-value* parameter to the matching call to process-lock, or else an error is signaled.

sys:lock-timeout (error) *Condition*

This condition is signaled when **process-lock** waits longer than the specified timeout.

It is a good idea to use **unwind-protect** to make sure that you unlock any lock that you seize. For example, if you write

```
(unwind-protect
    (progn (process-lock lock-3)
           (function-1)
           (function-2))
    (process-unlock lock-3))
```

then even if **function-1** or **function-2** does a **throw**, **lock-3** will get unlocked correctly. Particular programs that use locks often define special forms that package this **unwind-protect** up into a convenient stylistic device.

A higher level locking construct is **with-lock**:

with-lock (*lock* &key *norecursive*) *body...* *Macro*

Executes the *body* with *lock* locked. *lock* should actually be an expression whose value would be the status of the lock; it is used inside **locf** to get a locative pointer with which the locking and unlocking are done.

It is OK for one process to lock a lock multiple times, recursively, using **with-lock**, provided *norecursive* is not **nil**.

norecursive should be literally **t** or **nil**; it is not evaluated. If it is **t**, this call to **with-lock** signals an error if the lock is already locked by the running process.

A lower level construct which can be used to implement atomic operations, and is used in the implementation of **process-lock**, is **store-conditional**.

store-conditional *location oldvalue newvalue*

This stores *newvalue* into *location* iff *location* currently contains *oldvalue*. The value is **t** iff the cell was changed.

If *location* is a list, the cdr of the list is tested and stored in. This is in accord with the general principle of how to access the contents of a locative properly, and makes **(store-conditional (locf (cdr x)) ...)** work.

An even lower-level construct is the subprimitive **%store-conditional**, which is like **store-conditional** with no error checking, but is faster.

29.2.1 Process Queues

A process queue is a kind of lock which can record several processes which are waiting for the lock and grant them the lock in the order that they requested it. The queue has a fixed size. If the number of processes waiting remains less than that size then they will all get the lock in the order of requests. If too many processes are waiting then the order of requesting is not remembered for the extra ones, but proper interlocking is still maintained.

si:make-process-queue *name size*

Makes and returns a process queue object named *name*, able to record *size* processes. The count of *size* includes the process that owns the lock.

si:process-enqueue *process-queue* &optional *lock-value who-state*

Attempts to lock *process-queue* on behalf of *lock-value*. If *lock-value* is nil then the locking is done on behalf of current-process.

If the queue is locked, then *lock-value* or the current process is put on the queue. Then this function waits for that lock value to reach the front of the queue. When it does so, the lock has been granted, and the function returns. The lock is now locked in the name of *lock-value* or the current process, until si:process-dequeue is used to unlock it.

who-state appears in the who line during the wait. It defaults to "Lock".

si:process-deqeueue *process-queue* &optional *lock-value*

Unlocks process-queue. *lock-value* (which defaults to the current process) must be the value which now owns the lock on the queue, or an error occurs. The next process or other object on the queue is granted the lock and its call to si:process-enqueue will therefore return.

si:reset-process-queue *process-queue*

Unlocks the queue and clears out the list of things waiting to lock it.

si:process-queue-locker *process-queue*

Returns the object in whose name the queue is currently locked, or nil if it is not now locked.

29.3 Creating a Process

There are two ways of creating a process. One is to create a permanent process which you will hold on to and manipulate as desired. The other way is to say simply, "call this function on these arguments in another process, and don't bother waiting for the result." In the latter case you never actually use the process itself as an object.

make-process *name* &key ...

Creates and returns a process named *name*. The process is not capable of running until it has been reset or preset in order to initialize the state of its computation.

Usually you do not need to specify any of the keyword arguments. The following keyword arguments are allowed:

:simple-p	Specifying t here gives you a simple process (see page 695).
:flavor	Specifies the flavor of process to be created. See section 29.5, page 695, for a list of all the flavors of process supplied by the system.
:stack-group	Specifies the stack group the process is to use. If this option is not specified, a stack group is created according to the relevant options below.
:warm-boot-action	What to do with the process when the machine is booted. See page 693.
:quantum	See page 692.
:priority	See page 693.
:run-reasons	Lets you supply an initial run reason. The default is **nil**.
:arrest-reasons	Lets you supply an initial arrest reason. The default is **nil**.
:sg-area	The area in which to create the stack group. The default is the value of **default-cons-area**.
:regular-pdl-area :special-pdl-area :regular-pdl-size :special-pdl-size	These are passed on to **make-stack-group**, page 259.
:swap-sv-on-call-out :swap-sv-of-sg-that-calls-me :trap-enable	Specify those attributes of the stack group. You don't want to use these.

If you specify :flavor, there can be additional options provided by that flavor.

The following functions allow you to call a function and have its execution happen asynchronously in another process. This can be used either as a simple way to start up a process that will run "forever", or as a way to make something happen without having to wait for it to complete. When the function returns, the process is returned to a pool of free processes for reuse. The only difference between these three functions is in what happens if the machine is booted while the process is still active.

Normally the function to be run should not do any I/O to the terminal, as it does not have a window and terminal I/O will cause it to make a notification and wait for user attention. Refer to section 13.5, page 264 for a discussion of the issues.

process-run-function *name-or-options function* &rest *args*

Creates a process, presets it to apply *function* to *args*, and starts it running. The value returned is the new process. The process is killed if *function* returns; by default, it is also killed if it is reset. Example:

```
(defun background-print (file)
  (process-run-function "Print" 'hardcopy-file file))
```

creates a background process that prints the specified file.

name-or-options can be either a string specifying a name for the process or a list of alternating keywords and values that can specify the name and various other parameters.

:name	This keyword should be followed by a string which specifies the name of the process. The default is "Anonymous".
:restart-after-reset	This keyword says what to do to the process if it is reset. nil means the process should be killed; anything else means the process should be restarted. nil is the default.
:warm-boot-action	What to do with the process when the machine is booted. See page 693.
:restart-after-boot	This is a simpler way of saying what to do with the process when the machine is booted. If the :warm-boot-action keyword is not supplied or its value is nil, then this keyword's value is used instead. nil means the process should be killed; anything else means the process should be restarted. nil is the default.
:quantum	See page 692.
:priority	See page 693.

process-run-restartable-function *name-or-keywords function* &rest *args*

This is the same as process-run-function except that the default is that the process will be restarted if reset or after a warm boot. You can get the same effect by using process-run-function with appropriate keywords.

29.4 Process Generic Operations

These are the operations that are defined on all flavors of process. Certain process flavors may define additional operations. Not all possible operations are listed here, only those of interest to the user.

29.4.1 Process Attributes

:name *Operation on* si:process

Returns the name of the process, which was the first argument to make-process or process-run-function when the process was created. The name is a string that appears in the printed-representation of the process, stands for the process in the who-line and the peek display, etc.

:stack-group *Operation on* si:process

Returns the stack group currently executing on behalf of this process. This can be different from the initial-stack-group if the process contains several stack groups that coroutine among themselves, or if the process is in the error-handler, which runs in its own stack group.

Note that the stack-group of a *simple* process (see page 695) is not a stack group at all, but a function.

:initial-stack-group *Operation on* si:process

Returns the stack group the initial-function is called in when the process starts up or is reset.

:initial-form *Operation on* si:process

Returns the initial "form" of the process. This isn't really a Lisp form; it is a cons whose car is the initial-function and whose cdr is the list of arguments to which that function is applied when the process starts up or is reset.

In a simple process (see page 695), the initial form is a list of one element, the process's function.

To change the initial form, use the :preset operation (see page 694).

:wait-function *Operation on* si:process

Returns the process's current wait-function, which is the predicate used by the scheduler to determine if the process is runnable. This is #'true if the process is running, and #'false if the process has no current computation (for instance, if it has just been created, its initial function has returned) or if the program has decided to wait indefinitely. The wait-function of a flushed process is si:flushed-process, a function equivalent to #'false but recognizably distinct.

:wait-argument-list *Operation on* si:process

Returns the arguments to the process's current wait-function. This is frequently the &rest argument to process-wait in the process's stack. The system always uses it in a safe manner, i.e. it forgets about it before process-wait returns.

:whostate *Operation on* si:process

Returns a string that is the state of the process to go in the who-line at the bottom of the screen. This is "run" if the process is running or trying to run; otherwise, it is the reason why the process is waiting. If the process is stopped, then this whostate string is ignored and the who-line displays arrest if the process is arrested or stop if the process has no run reasons.

:quantum *Operation on* si:process
:set-quantum *60ths* *Operation on* si:process

Respectively return and change the number of 60ths of a second this process is allowed to run without waiting before the scheduler will run someone else. The quantum defaults to 1 second.

:quantum-remaining *Operation on* si:process
Returns the amount of time remaining for this process to run, in 60ths of a second.

:priority *Operation on* si:process
:set-priority *priority-number* *Operation on* si:process
Respectively return and change the priority of this process. The larger the number, the more this process gets to run. Within a priority level the scheduler runs all runnable processes in a round-robin fashion. Regardless of priority a process will not run for more than its quantum. The default priority is 0, and no normal process uses other than 0.

:warm-boot-action *Operation on* si:process
:set-warm-boot-action *action* *Operation on* si:process
Respectively return and change the process's warm-boot-action, which controls what happens if the machine is booted while this process is active. (Contrary to the name, this applies to both cold and warm booting.) This can be nil or :flush, which means to *flush* the process (see page 695), or can be a function to call. The default is si:process-warm-boot-delayed-restart, which resets the process, causing it to start over at its initial function. You can also use si:process-warm-boot-reset, which throws out of the process' computation and kills the process, or si:process-warm-boot-restart, which is like the default but restarts the process at an earlier stage of system reinitialization. This is used for processes like the keyboard process and chaos background process, which are needed for reinitialization itself.

:simple-p *Operation on* si:process
Returns nil for a normal process, t for a simple process. See page 695.

:idle-time *Operation on* si:process
Returns the time in seconds since this process last ran, or nil if it has never run.

:total-run-time *Operation on* si:process
Returns the amount of time this process has run, in 60ths of a second. This includes cpu time and disk wait time.

:disk-wait-time *Operation on* si:process
Returns the amount of time this process has spent waiting for disk I/O, in 60ths of a second.

:cpu-time *Operation on* si:process
Returns the amount of time this process has spent actually executing instructions, in 60ths of a second.

:percent-utilization *Operation on* si:process
Returns the fraction of the machine's time this process has been using recently, as a percentage (a number between 0 and 100.0). The value is a weighted average giving more weight to more recent history.

:reset-meters *Operation on* si:process
Resets the run-time counters of the process to zero.

29.4.2 Run and Arrest Reasons

:run-reasons *Operation on* si:process
Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

:run-reason *object* *Operation on* si:process
Adds *object* to the process's run reasons. This can activate the process.

:revoke-run-reason *object* *Operation on* si:process
Removes *object* from the process's run reasons. This can stop the process.

:arrest-reasons *Operation on* si:process
Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

:arrest-reason *object* *Operation on* si:process
Adds *object* to the process's arrest reasons. This can stop the process.

:revoke-arrest-reason *object* *Operation on* si:process
Removes *object* from the process's arrest reasons. This can activate the process.

:active-p *Operation on* si:process
:runnable-p *Operation on* si:process
These two operations are the same. t is returned if the process is active, i.e. it can run if its wait-function allows. nil is returned if the process is stopped.

29.4.3 Bashing the Process

:preset *function* &rest *args* *Operation on* si:process
Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that any computation occuring in it is terminated and it begins anew by applying *function* to *args*. A :preset operation on a stopped process returns immediately, but does not activate the process; hence the process will not really apply *function* to *args* until it is activated later.

:reset &optional *no-unwind kill* *Operation on* si:process
Forces the process to throw out of its present computation and apply its initial function to its initial arguments, when it next runs. The throwing out is skipped if the process has no present computation (e.g. it was just created), or if the *no-unwind* option so specifies. The possible values for *no-unwind* are:

:unless-current	
nil	Unwind unless the stack group to be unwound is the one currently executing, or belongs to the current process.

:always	Unwind in all cases. This may cause the operation to throw through its caller instead of returning.
t	Never unwind.

If *kill* is t, the process is to be killed after unwinding it. This is for internal use by the :kill operation only.

A :reset operation on a stopped process returns immediately, but does not activate the process; hence the process will not really get reset until it is activated later.

:flush *Operation on* si:process

Forces the process to wait forever. A process may not :flush itself. Flushing a process is different from stopping it, in that it is still active and hence if it is reset or preset it will start running again.

A flushed process can be recognized because its wait-function is si:flushed-process. If a process belonging to a window is flushed, exposing or selecting the window resets the process.

:kill *Operation on* si:process

Gets rid of the process. It is reset, stopped, and removed from sys:all-processes.

:interrupt *function* &rest *args* *Operation on* si:process

Forces the process to apply *function* to *args*. When *function* returns, the process continues the interrupted computation. If the process is waiting, it wakes up, calls *function*, then waits again when *function* returns.

If the process is stopped it does not apply *function* to *args* immediately, but will later when it is activated. Normally the :interrupt operation returns immediately, but if the process's stack group is in an unusual internal state :interrupt may have to wait for it to get out of that state.

29.5 Process Flavors

These are the flavors of process provided by the system. It is possible for users to define additional flavors of their own.

si:process *Flavor*

This is the standard default kind of process.

si:simple-process *Flavor*

A simple process is not a process in the conventional sense. It has no stack group of its own; instead of having a stack group that gets resumed when it is time for the process to run, it has a function that gets called when it is time for the process to run. When the wait-function of a simple process becomes true, and the scheduler notices it, the simple process's function is called in the scheduler's own stack group. Since a simple process does not have any stack group of its own, it can't save control state in between calls; any state that it saves must be saved in data structure.

The only advantage of simple processes over normal processes is that they use up less system overhead, since they can be scheduled without the cost of resuming stack-groups. They are intended as a special, efficient mechanism for certain purposes. For example, packets received from the Chaosnet are examined and distributed to the proper receiver by a simple process that wakes up whenever there are any packets in the input buffer. However, they are harder to use, because you can't save state information across scheduling. That is, when the simple process is ready to wait again, it must return; it can't call **process-wait** and continue to do something else later. In fact, it is an error to call **process-wait** from inside a simple process. Another drawback to simple processes is that if the function signals an error, the scheduler itself is broken and multiprocessing stops; this situation can be repaired only by aborting, which blasts the process. Also, when a simple process is run, no other process is scheduled until it chooses to return; so simple processes should never run for a long time without returning.

Asking for the stack group of a simple process does not signal an error, but returns the process's function instead.

Since a simple process cannot call **process-wait**, it needs some other way to specify its wait-function. To set the wait-function of a simple process, use **si:set-process-wait** (see below). So, when a simple process wants to wait for a condition, it should call **si:set-process-wait** to specify the condition, then return.

si:set-process-wait *simple-process wait-function wait-argument-list*
Sets the *wait-function* and *wait-argument-list* of *simple-process*. See the description of the **si:simple-process** flavor (above) for more information.

29.6 Other Process Functions

process-enable *process*
Activates *process* by revoking all its run and arrest reasons, then giving it a run reason of **:enable**.

process-reset-and-enable *process*
Resets *process*, then enables it.

process-disable *process*
Stops *process* by revoking all its run reasons. Also revokes all its arrest reasons.

The remaining functions in this section are obsolete, since they simply duplicate what can be done by sending a message. They are documented here because their names are in the **global** package.

process-preset *process function* &rest *args*
Sends a **:preset** message.

process-reset *process*
: Sends a :reset message.

process-name *process*
: Gets the name of a process, like the :name operation.

process-stack-group *process*
: Gets the current stack group of a process, like the :stack-group operation.

process-initial-stack-group *process*
: Gets the initial stack group of a process, like the :initial-stack-group operation.

process-initial-form *process*
: Gets the initial form of a process, like the :initial-form operation.

process-wait-function *process*
: Gets the current wait-function of a process, like the :wait-function operation.

process-wait-argument-list *p*
: Gets the arguments to the current wait-function of a process, like the :wait-argument-list operation.

process-whostate *p*
: Gets the current who-line state string of a process, like the :whostate operation.

30. Errors and Debugging

The first portion of this chapter explains how programs can handle errors, by means of condition handlers. It also explains how a program can signal an error if it detects something it doesn't like.

The second explains how users can handle errors, by means of an interactive debugger; that is, it explains how to recover if you do something wrong. A new user of the Lisp Machine, or someone who just wants to know how to deal with errors and not how to cause them, should ignore the first sections and skip ahead to section 30.7, page 726.

The remaining sections describe some other debugging facilities. Anyone who is going to be writing programs for the Lisp Machine should familiarize himself with these.

The *trace* facility provides the ability to perform certain actions at the time a function is called or at the time it returns. The actions may be simple typeout, or more sophisticated debugging functions.

The *advise* facility is a somewhat similar facility for modifying the behavior of a function.

The *breakon* facility allows you to cause the debugger to be entered when a certain function is called. You can then use the debugger's stepping commands to step to the next function call or return.

The *step* facility allows the evaluation of a form to be intercepted at every step so that the user may examine just what is happening throughout the execution of the form. Stepping works only on interpreted code.

The *MAR* facility provides the ability to cause a trap on any memory reference to a word (or a set of words) in memory. If something is getting clobbered by agents unknown, this can help track down the source of the clobberage.

30.1 Conditions

Programmers often want to control what action is taken by their programs when errors or other exceptional situations occur. Usually different situations are handled in different ways, and in order to express what kind of handling each situation should have, each situation must have an associated name. In Zetalisp, noteworthy events are represented by objects called *condition instances*. When an event occurs, a condition instance is created; it is then *signaled*, and a *handler* for that condition may be invoked.

When a condition is signaled, the system (essentially) searches up the stack of nested function invocations looking for a handler established to handle that condition. The handler is a function that gets called to deal with the condition. The condition mechanism itself is just a convenient way for finding an appropriate handler function for a particular exceptional situation.

When a condition is signaled, a *condition instance* is created to represent the event and hold information about it. This information includes *condition names* then classify the condition and any other data that is likely to be of interest to condition handlers. A condition instance is immutable once it has been created. Some conditions are *errors*, which means that the debugger is invoked if they are signaled and not handled.

Condition instances are flavor instances. The flavor **condition** is the base flavor from which all flavors of condition are built. Several operations that are defined on condition instances are described below. The flavor **error**, which is built on **condition**, is the base flavor for all kinds of conditions which are errors.

A *condition name* is a symbol then is used to identify a category of conditions. Each condition instance possesses one or more condition names. Each condition handler specifies one or more condition names that it should apply to. A handler applies to a condition if they have any condition names in common. This is the sole purpose of condition names: to match condition instances with their handlers. The meaning of every condition name signaled by the system is described in this manual. The condition name index is a directory for them. Conditions that are errors possess the condition name **error**.

In PL/I, CLU, ADA and most other systems that provide named conditions, each condition has only one name. That is to say, the categories identified by condition names are disjoint. In Zetalisp, each condition instance can have multiple condition names, which means that the categories identified by condition names can overlap and be subdivided.

For example, among the condition names defined by the system are **condition**, **error**, **sys:arithmetic-error**, **sys:floating-exponent-underflow** and **sys:divide-by-zero**. **condition** is a condition name that all condition instances possess. **error** identifies the category of conditions that are considered errors. **sys:arithmetic-error** identifies the category of errors that pertain to arithmetic operations. **sys:floating-exponent-underflow** and **sys:divide-by-zero** are the most specific level of categorization. So, the condition signaled when you evaluate **(* 1s-30 1s-30 1s-30 1s-30)** possesses condition names **sys:floating-exponent-underflow**, **sys:arithmetic-error**, **error** and **condition**, while the one signaled if you evaluate **(// 1 0)** possesses condition names **sys:divide-by-zero**, **sys:arithmetic-error**, **error** and **condition**. In this example, the categories fall into a strict hierarchy, but this does not need to be the case.

Condition names are documented throughout the manual, with definitions like this:

sys:divide-by-zero (sys:arithmetic-error error) *Condition*

The condition name **sys:divide-by-zero** is always accompanied by **sys:arithmetic-error** and **error** (that is, it categorizes a subset of those categories). The presence of **error** implies that all **sys:divide-by-zero** conditions are errors.

The condition instance also records additional information about the event. For example, the condition instance signaled by dividing by zero handles the **:function** operation by returning the function that did the division (it might be **truncate**, **floor**, **ceiling** or **round**, as well as **//**). In general, for each condition name there are conventions saying what additional information is provided and what operations to use to obtain it.

The flavor of the condition instance is always one of the condition names, and so are its component flavors (with a few exceptions; si:vanilla-flavor and some other flavor components are omitted, since they are not useful categories for condition handlers to specify). In our example, the flavor of the condition is sys:arithmetic-error, and its components include error and condition. Condition names require new flavors only when they require significantly different handling by the error system; you will understand in detail after finishing this section.

condition-typep *condition-instance condition-name*

Returns t if *condition-instance* possesses condition name *condition-name*. *condition-name* can also be a combination of condition names using and, or and not; then the condition tested for is a boolean combination of the presence or absence of various condition names. Example:

```
(condition-typep error 'fs:file-not-found)
(condition-typep error
   '(or fs:file-not-found fs:directory-not-found))
```

errorp *object*

Returns t if *object* is a condition instance and its flavor incorporates error. This is normally equivalent to (typep *object* 'error). Some functions such as open optionally return the condition instance rather than signaling it, if an error occurs. errorp is useful in testing the value returned.

:condition-names *Operation on* condition

Returns a list of all the condition names possesses by this condition instance.

30.2 Handling Conditions

A condition handler is a function that is associated with certain condition names (categories of conditions). The variable eh:condition-handlers contains a list of the handlers that are current; handlers are established using macros which bind this variable. When a condition is signaled, this list is scanned and all the handlers which apply are called, one by one, until one of the handlers either throws or returns non-nil.

Since each new handler is pushed onto the front of eh:condition-handlers, the innermost-established handler gets the first chance to handle the condition. When the handler is run, eh:condition-handlers is bound so that the running handler (and all the ones that were established farther in) are not in effect. This avoids the danger of infinite recursion due to an error in a handler invoking the same handler.

One thing a handler can do is throw to a tag. Often the catch for this tag is right next to the place where the handler is established, but this does not have to be so. A simple handler that applies to all errors and just throws to a tag is established using ignore-errors.

ignore-errors *body...* *Macro*

An error within the execution of *body* causes control to return from the ignore-errors form. In this case, the values are nil, t. If there is no error inside *body*, the first value is that of the last form in the *body* and the second is nil.

Errors whose condition instances return true for the :dangerous-condition-p operation are not handled. These include such things as running out of virtual memory.

A handler can also signal another condition. For example, signaling **sys:abort** has the effect of pretending that the user typed the **Abort** key. The following function creates a handler which signals **sys:abort**.

si:eval-abort-trivial-errors *form*

Evaluates *form* with a condition handler for many common error conditions such as :wrong-type-argument, :unbound-variable and :unclaimed-message. The handler asks the user whether to allow the debugger to be entered. If the user says 'no', the handler signals the sys:abort condition. If the user says 'yes', the handler does not handle the condition, allowing the debugger to do so.

In some cases the handler attempts to determine whether the incorrect variable, operation, or argument appeared in *form*; if it did not, the debugger is always allowed to run. The assumption is that *form* was typed in by the user, and the intention is to distinguish trivial mistakes from program bugs.

The handler can also ask to proceed from the condition. This is done by returning a non-nil value. See the section on proceeding, page 717, for more information.

The handler can also decline to handle the condition, by returning **nil**. Then the next applicable handler is called, and so on until either some handler does handle the condition or there are no more handlers.

The handler function is called in the environment where the condition was signaled, and in the same stack group. All special variables have the values they had at the place where the signaling was done, and all catch tags that were available at the point of signaling may be thrown to.

The handler receives the condition instance as its first argument. When establishing the handler, you can also provide additional arguments to pass to the handler when it is called. This allows the same function to be used in varying circumstances.

The fundamental means of establishing a condition handler is the macro **condition-bind**.

condition-bind (*handlers...*) *body...* *Macro*
condition-bind-default (*handlers...*) *body...* *Macro*

A condition-bind form looks like this:

```
(condition-bind ((conditions handler-form additional-arg-forms...)
                 (conditions handler-form additional-arg-forms...))
  body...)
```

The purpose is to execute *body* with one or more condition handlers established.

Each list of conditions and handler-form establishes one handler. *conditions* is a condition name or a list of condition names to which the handler should apply. It is *not* evaluated. *handler-form* is evaluated to produce the function that is the actual handler. The *additional-arg-forms* are evaluated, on entry to the condition-bind, to produce additional arguments that are passed to the handler function when it is called. The arguments to the handler function are the condition instance being signaled, followed by the values of any *additional-arg-forms*.

conditions can be **nil**; then the handler applies to all conditions that are signaled. In this case it is up to the handler function to decide whether to do anything. It is important for the handler to refrain from handling certain conditions that are used for debugging, such as **break** and **eh:call-trap**. The **:debugging-condition-p** operation on condition instances returns non-**nil** for these conditions. Certain other conditions such as **sys:virtual-memory-overflow** should be handled only with great care. The **:dangerous-condition-p** operation returns non-nil for these conditions. Example:

```
(condition-bind ((nil 'myhandler "it happened here" 45))
  (catch 'x
    ...))

(defun myhandler (condition string value)
  (unless (or (condition-typep condition 'fs:file-error)
              (send condition :dangerous-condition-p)
              (send condition :debugging-condition-p))
    (format error-output "~&~A:~%~A~%" string condition)
    (throw 'x value)))
```

myhandler declines to handle file errors, and all debugging conditions and dangerous errors. For all other conditions, it prints the string specified in the condition bind and throws to the tag x the value specified there (45).

condition-bind-default is like **condition-bind** but establishes a *default handler* instead of an ordinary handler. Default handlers work like ordinary handlers, but they are tried in a different order: first all the applicable ordinary handlers are given a chance to handle the condition, and then the default handlers get their chance. A more flexible way of doing things like this is described under **signal-condition** (page 715).

Condition handlers that simply throw to the function that established them are very common, so there are special constructs provided for defining them.

condition-case (*variables...*) *body-form clauses...* *Macro*

```
(condition-case (variable)
    body-form
  (condition-names forms...)
  (condition-names forms...)
  ...)
```

body-form is executed with a condition handler established that will throw back to the **condition-case** if any of the specified condition names is signaled.

Each list starting with some condition names is a *clause*, and specifies what to do if one of those condition names is signaled. *condition-names* is either a condition name or a list of condition names; it is not evaluated.

Once the handler per se has done the throw, the clauses are tested in order until one is found that applies. This is almost like a **selectq**, except that the signaled condition can have several condition names, so the first clause that matches any of them gets to run. The forms in the clause are executed with *variable* bound to the condition instance that was signaled. The values of the last form in the clause are returned from the **condition-case** form.

If none of the specified conditions is signaled during the execution of *body-form* (or if other handlers, established within *body-form*, handle them) then the values of *body-form* are returned from the **condition-case** form.

variable may be omitted if it is not used.

It is also possible to have a clause starting with :no-error in place of a condition name. This clause is executed if *body-form* finishes normally. Instead of just one *variable* there can be several variables; during the execution of the :no-error clause, these are bound to the values returned by *body-form*. The values of the last form in the clause become the values of the **condition-case** form.

Here is an example:

```
(condition-case ()
    (print foo)
  (error (format t " <<Error in printing>>")))
```

condition-call (*variables...*) *body-form clauses...* *Macro*

condition-call is an extension of **condition-case** that allows you to give each clause an arbitrary conditional expression instead of just a list of condition names. It looks like this:

```
(condition-call (variables...)
    body-form
  (test forms...)
  (test forms...)
  ...)
```

The difference between this and **condition-case** is the *test* in each clause. The clauses in a **condition-call** resemble the clauses of a **cond** rather than those of a **selectq**.

When a condition is signaled, each *test* is executed while still within the environment of the signaling (that is, within the actual handler function). The condition instance can be found in the first *variable*. If any *test* returns non-nil, then the handler throws to the **condition-call** and the corresponding clause's *forms* are executed. If every *test* returns **nil**, the condition is not handled by this handler.

In fact, each *test* is computed a second time after the throw has occurred in order to decide which clause to execute. The code for the *test* is copied in two different places, once into the handler function to decide whether to throw, and once in a **cond** which follows the catch.

The last clause can be a :no-error clause just as in **condition-case**. It is executed if the body returns without error. The values returned by the body are stored in the *variables*. The values of the last form in the :no-error clause are returned by the **condition-call**.

Only the first of *variables* is used if there is no :no-error clause. The *variables* may be omitted entirely in the unlikely event that none is used. Example:

```
(condition-call (instance)
    (do-it)
  ((condition-typep instance
     '(and fs:file-error (not fs:no-more-room)))
   (compute-what-to-return)))
```

The condition name **fs:no-more-room** is a subcategory of **fs:file-error**; therefore, this handles all file errors *except* for **fs:no-more-room**.

Each of the four condition handler establishing constructs has a conditional version that decides at run time whether to establish the handlers.

condition-bind-if *cond-form* (*handlers...*) *body...* *Macro*

```
(condition-bind-if cond-form
                   ((conditions handler-form additional-arg-forms...)
                    (conditions handler-form additional-arg-forms...))
  body...)
```

begins by executing *cond-form*. If it returns non-nil, then all proceeds as for a regular condition-bind. If *cond-form* returns nil, then the *body* is still executed but without the condition handler.

condition-case-if *cond-form* (*variables...*) *body-form* *clauses...* *Macro*

```
(condition-case-if cond-form (variables...)
    body-form
  (condition-names forms...)
  (condition-names forms...)
  ...)
```

begins by executing *cond-form*. If it returns non-nil, then all proceeds as for a regular condition-case. If *cond-form* returns nil, then the *body-form* is still executed but without the condition handler. *body-form*'s values are returned, or, if there is a :no-error clause, it is executed and its values returned.

condition-call-if *cond-form* ([*variable*]) *body-form* *clauses...* *Macro*

```
(condition-call-if cond-form (variables...)
    body-form
  (test forms...)
  (test forms...)
  ...)
```

begins by executing *cond-form*. If it returns non-nil, then everything proceeds as for a regular condition-call. If *cond-form* returns nil, then the *body-form* is still executed but without the condition handler. In that case, *body-form*'s values are returned, or, if there is a :no-error clause, it is executed and its values returned.

condition-bind-default-if *cond-form* (*handlers...*) *body...* *Macro*

This is used just like condition-bind-if, but establishes a default handler instead of an ordinary handler.

eh:condition-handlers *Variable*

This is the list of established condition handlers. Each element looks like this:

(*condition-names function additional-arg-values*...)

condition-names is a condition name or a list of condition names, or nil which means all conditions.

function is the actual handler function.

additional-arg-values are additional arguments to be passed to the *function* when it is called. *function*'s first argument is always the condition instance.

Both the links of the value of eh:condition-handlers and the elements are usually created with with-stack-list, so copy them if you want to save them for any period of time.

eh:condition-default-handlers *Variable*

This is the list of established default condition handlers. The data format is the same as that of eh:condition-handlers.

30.3 Standard Condition Flavors

condition *Flavor*

The flavor condition is the base flavor of all conditions, and provides default definitions for all the operations described in this chapter.

condition incorporates si:property-list-mixin, which defines operations :get and :plist. Each property name on the property list is also an operation name, so that sending the :foo message is equivalent to (send instance :get :foo). In addition, (send instance :set :foo *value*) is equivalent to (send instance :set :get :foo *value*).

condition also provides two instance variables, eh:format-string and eh:format-args. condition's method for the the :report operation passes these to format to print the error message.

error *Flavor*

The flavor error makes a condition an error condition. errorp returns t for such conditions, and the debugger is entered if they are signaled and not otherwise handled.

sys:no-action-mixin *Flavor*

This mixin provides a definition of the proceed type :no-action.

sys:proceed-with-value-mixin *Flavor*

This mixin provides a definition of the proceed type :new-value.

ferror *Flavor*

This flavor is a mixture of error, sys:no-action-mixin and sys:proceed-with-value-mixin. It is the flavor used by default by the functions **ferror** and **cerror**, and is often convenient for users to instantiate.

sys:warning *Flavor*

This flavor is a mixture of sys:no-action-mixin and condition.

sys:bad-array-mixin *Flavor*

This mixin provides a definition of the proceed type :new-array.

30.4 Condition Operations

Every condition instance can be asked to print an *error message* which describes the circumstances that led to the signaling of the condition. The easiest way to print one is to print the condition instance without escaping (princ, or format operation ~A). This actually uses the :report operation, which implements the printing of an error message. When a condition instance is printed with escaping, it uses the #c syntax so that it can be read back in. This is done using si:print-readably-mixin, page 446.

:report *stream* *Operation on* condition

Prints on *stream* the condition's error message, a description of the circumstances for which the condition instance was signaled. The output should neither start nor end with a carriage return.

If you are defining a new flavor of condition and wish to change the way the error message is printed, this is the operation to redefine. All others use this one.

:report-string *Operation on* condition

Returns a string containing the text that the :report operation would print.

Operations provided specifically for condition handlers to use:

:dangerous-condition-p *Operation on* condition

Returns **t** if the condition instance is one of those that indicate events that are considered extremely dangerous, such as running out of memory. Handlers that normally handle all conditions might want to make an exception for these.

:debugging-condition-p *Operation on* condition

Returns **t** if the condition instance is one of those that are signaled as part of debugging, such as break, which is signaled when you type Meta-Break. Although these conditions normally enter the debugger, they are not errors; this serves to prevent most condition handlers from handling them. But any condition handler which is written to handle *all* conditions should probably make a specific exception for these.

See also the operations :proceed-types and :proceed-type-p, which have to do with proceeding (page 717).

30.4.1 Condition Operations for the Debugger

Some operations are intended for the debugger to use. They are documented because some flavors of condition redefine them so as to cause the debugger to behave differently. This section is of interest only to advanced users.

:print-error-message *stack-group brief-flag stream* — *Operation on* condition

This operation is used by the debugger to print a complete error message. This is done primarily using the :report operation.

Certain flavors of condition define a :after :print-error-message method which, when *brief-flag* is nil, prints additional helpful information which is not part of the error message per se. Often this requires access to the stack group in addition to the data in the condition instance. The method can assume that if *brief-flag* is nil then *stack-group* is not the one which is executing.

For example, the :print-error-message method of the condition signaled when you call an undefined function checks for the case of calling a function such as **bind** that is meaningful only in compiled code; if that is what happened, it searches the stack to look for the name of the function in which the call appears. This is information that is not considered crucial to the error itself, and is therefore not recorded in the condition instance.

:maybe-clear-input *stream* — *Operation on* condition

This operation is used on entry to the debugger to discard input. Certain condition flavors, used by stepping redefine this operation to do nothing, so that input is not discarded.

:bug-report-recipient-system — *Operation on* condition

The value returned by this operation is used to determine what address to mail bug reports to, when the debugger Control-M command is used. By default, it returns "LISPM". The value is passed to the function **bug**.

:bug-report-description *stream* &optional *numeric-arg* — *Operation on* condition

This operation is used by the Control-M command to print on *stream* the information that should go in the bug report. *numeric-arg* is the numeric argument, if any, that the user gave to the Control-M command.

:find-current-frame *stack-group* — *Operation on* condition

Returns the stack indices of the stack frames that the debugger should operate on.

The first value is the frame "at which the error occurred." This is not the innermost stack frame; it is outside the calls to such functions as **ferror** and **signal-condition** which were used to signal the error.

The second value is the initial value for the debugger command loop's current frame.

The third value is the innermost frame that the debugger should be willing to let the user see. By default this is the innermost active frame, but it is safe to use an open but not active frame within it.

The fourth value, if non-nil, tells the debugger to consider the innermost frame to be "interesting". Normally, frames that are part of the interpreter (calls to si:eval1, si:apply-lambda, prog, cond, etc.) are considered uninteresting.

This is a flavor operation so that certain flavors of condition can redefine it.

:debugger-command-loop *Operation on* condition
stack-group &optional *error-object*

Enters the debugger command loop. The initial error message and backtrace have already been printed. This message is sent in an error handler stack group; *stack-group* is the stack group in which the condition was signaled. *error-object* is the condition object which was signaled; it defaults to the one the message is sent to.

This operation uses :or method combination (see section 21.11, page 433). Some condition flavors add methods that perform some other sort of processing or enter a different command loop. For example, unbound variable errors look for look-alike symbols in other packages at this point. If the added method returns nil, the original method that enters the usual debugger command loop is called.

30.5 Signaling Conditions

Signaling a condition has two steps, creating a condition instance and signaling the instance. There are convenience interface functions that combine the two steps. You can also do them separately. If you just want to signal an error and do not want to worry much about condition handling, the function ferror is all you need to know.

30.5.1 Convenience Functions for Signaling

ferror &rest *make-condition-arguments*

Creates a condition instance using make-condition and then signals it with signal-condition, specifying no local proceed types, and with t as the *use-debugger* argument so the debugger is always entered if the condition is not otherwise handled.

The first argument to ferror is always a signal name (often nil). The second argument is usually a format string and the remaining arguments are additional arguments for format; but this is under the control of the definition of the signal name. Example:

```
(ferror 'math:singular-matrix
        "The matrix ~S cannot be inverted." matrix)
```

For compatibility with the Symbolics system, if the first argument to ferror is a string, then a signal name of nil is assumed. The arguments to ferror are passed on to make-condition with an additional nil preceding them.

If you prefer, you can use the formatted output functions (page 496) to generate the error message. Here is an example, though in a simple case like this using **format** is easier:

```
(ferror 'math:singular-matrix
  (format:outfmt
    "The matrix "
    (prin1 matrix)
    " cannot be inverted.")
  number)
```

In this case, arguments past the second one are not used for printing the error message, but the signal name may still expect them to be present so it can put them in the condition instance.

cerror *proceed-type ignore signal-name* &rest *signal-name-arguments*

Creates a condition instance, by passing the *signal-name* and *signal-name-arguments* to **make-condition**, and then signals it. If *proceed-type* is non-**nil** then it is provided to **signal-condition** as a proceed type. For compatibility with old uses of **cerror**, if *proceed-type* is **t**, **:new-value** is provided as the proceed type. If *proceed-type* is **:yes**, **:no-action** is provided as the proceed type.

The second argument to **cerror** is not used and is present for historical compatibility. It may be given a new meaning in the future.

If a condition handler or the debugger decides to proceed, the second value it returns becomes the value of **cerror**.

Common Lisp defines another calling sequence for this function:

(cerror *continue-format-string error-format-string args...*)

This signals an error of flavor **eh:common-lisp-cerror**, which prints an error message using *error-format-string* and *args*. It allows one proceed type, whose documentation is *continue-format-string*, and which proceeds silently, returning **nil** from **cerror**. **cerror** can tell which calling sequence has been used and behaves accordingly.

warn *format-string* &rest *args*

Prints a warning on ***error-output*** by passing the args to **format**, starting on a fresh line, and then returns.

If ***break-on-warnings*** is non-**nil**, however, **warn** signals a procedable error, using the arguments to make an error message. If the user proceeds, **warn** simply returns.

break-on-warnings

If non-**nil**, **warn** signals an error rather than just printing a message.

check-type *place type-spec* [*description*] *Macro*
check-arg-type *place type-spec* [*description*] *Macro*

Signals a correctable error if the value of *place* does not fit the type *type-spec*. *place* is something that **setf** can store in. *type-spec* is a type specifier, a suitable second argument to **typep**, and is not evaluated (see section 2.3, page 14). A simple example is:

```
(check-type foo (integer 0 10))
```

This signals an error unless **(typep foo '(integer 0 10))**; that is, unless **foo**'s value is an integer between zero and ten, inclusive.

If an error is signaled, the error message contains the name of the variable or place where the erroneous value was found, and the erroneous value itself. An English description of the type of object that was wanted is computed automatically from the type specifier for use in the error message. For the commonly used type specifiers this computed description is adequate. If it is unsatisfactory in a particular case, you can specify *description*, which is used instead. In order to make the error message grammatical, *description* should start with an indefinite article.

The error signaled is of condition **sys:wrong-type-argument** (see page 61). The proceed type **:argument-value** is provided. If a handler proceeds using this proceed type, it should specify one additional argument; that value is stored into *place* with **setf**. The new value is then tested, and so on. **check-type** returns when a value passes the test.

check-arg-type is an older name for this macro.

check-arg *var-name predicate description* [*type-symbol*] *Macro*

check-arg is an obsolete variant of **check-type**. *predicate* is either a symbol which is predicate (a function of one argument) or a list which is a form. If it is a predicate, it is applied to the value of *var-name*, which is valid if the predicate returns non-**nil**. If it is a form, it is evaluated, and the value is valid of the form returns non-**nil**. The form ought to make use of *var-name*, but nothing checks this.

There is no way to compute an English description of valid values from *predicate*, so a *description* string must always be supplied.

type-symbol is a symbol that is used by condition handlers to determine what type of argument was expected. If *predicate* is a symbol, you may omit *type-symbol*, and *predicate* is used for that purpose as well. The use of the *type-symbol* is not really well-defined, and **check-type**, where a type specifier serves both purposes, is superior to **check-arg** for this reason.

Examples:

```
(check-arg foo stringp "a string")

(check-arg h
           (or (stringp h) (typep h 'fs:host))
           "a host name"
           fs:host)
```

Other functions that can be used to test for invalid values include **ecase** and **ccase** (page 66), which are error-checking versions of **selectq**, and **etypecase** and **ctypecase** (page 21), error-checking versions of **typecase**.

assert *test-form* [(*places*...) [*string args*...]] *Macro*

Signals an error if *test-form* evaluates to **nil**. The rest of the **assert** form is relevant only if the error happens.

First of all, the *places* are forms that can be stored into with **setf**, and which are used (presumably) in *test-form*. The reason that the *places* are specified again in the **assert** is so that the expanded code can arrange for the user to be able to specify a new value to be stored into any one of them when he proceeds from the error. When the error is signaled, one proceed-type is provided for each *place* that is given. The condition object has flavor **eh:failed-assertion**.

If the user does proceed with a new value in that fashion, the *test-form* is evaluated again, and the error repeats until the *test-form* comes out non-**nil**.

The *string* and *args* are used to print the error message. If they are omitted, **"Failed assertion"** is used. They are evaluated only when an error is signaled, and are evaluated again each time an error is signaled. **setf**'ing the *places* may also involve evaluation, which happens each time the user proceeds and sets one.

Example:
```
    (assert (neq (car a) (car b)) ((car a) (car b))
            "The CARS of A and B are EQ: ~S and ~S"
            (car a) (car b))
```
The *places* here are **(car a)** and **(car b)**. The *args* happen to be the same two forms, by not-exactly-coincidence; the current values of the *places* are often useful in the error message.

The remaining signaling functions are provided for compatibility only.

error &rest *make-condition-arguments*

error exists for compatibility with Maclisp and the Symbolics version of Zetalisp. It takes arguments in three patterns:
```
    (error string object [interrupt])
```
which is used in Maclisp, and
```
    (error condition-instance)
    (error flavor-name init-options...)
```
which are used by Symbolics. (In fact, the arguments to **error** are simply passed along to **make-condition** if they do not appear to fit the Maclisp pattern).

If the Maclisp argument pattern is not used then there is no difference between **error** and **ferror**.

cli:error *format-string* &rest *args*

The Common Lisp version of **error** signals an uncorrectable error whose error message is printed by passing *format-string* and *args* to **format**.

fsignal *format-string* &rest *format-args*

This function is for Symbolics compatibility only, and is equivalent to

```
        (cerror :no-action nil nil format-string format-args...)
```

signal *signal-name* &rest *remaining-make-condition-arguments*

The *signal-name* and *remaining-make-condition-arguments* are passed to make-condition, and the result is signaled with signal-condition.

If the *remaining-make-condition-arguments* are keyword arguments and :proceed-types is one of the keywords, the associated value is used as the list of proceed types. In particular, if *signal-name* is actually a condition instance, so that the remaining arguments will be ignored by make-condition, it works to specify the proceed types this way.

If the proceed types are not specified, a list of all the proceed types that the condition instance knows how to prompt the user about is used by default.

errset *form* [*flag*] *Macro*

Catches errors during the evaluation of *form*. If an error occurs, the usual error message is printed unless *flag* is nil. Then control is thrown and the errset-form returns nil. *flag* is evaluated first and is optional, defaulting to t. If no error occurs, the value of the errset-form is a list of one element, the value of *form*.

errset is an old, Maclisp construct, implemented much like condition-case. Many uses of errset or errset-like constructs really ought to be checking for more specific conditions instead.

catch-error *form* [*flag*] *Macro*

catch-error is a variant of errset. This construct catches errors during the evaluation of *form* and returns two values. If *form* returns normally, the first value is *form*'s first value and the second value is nil. If an error occurs, the usual error message is printed unless *flag* is nil, and then control is thrown out of the catch-error form, which returns two values, first nil and second a non-nil value that indicates the occurrence of an error. *flag* is evaluated before *form* and is optional, defaulting to t.

errset *Variable*

If this variable is non-nil, errset forms are not allowed to trap errors. The debugger is entered just as if there were no errset. This is intended mainly for debugging. The initial value of errset is nil.

err *Macro*

This is for Maclisp compatibility only and should not be used.

(err) is a dumb way to cause an error. If executed inside an errset, that errset returns nil, and no message is printed. Otherwise an error is signaled with error message just "ERROR>>".

(err *form*) evaluates *form* and causes the containing errset to return the result. If executed when not inside an errset, an error is signaled with *form*'s value printed as the error message.

(err *form flag*), which exists in Maclisp, is not supported.

30.5.2 Creating Condition Instances

You can create a condition instance quite satisfactorily with **make-instance** if you know which instance variables to initialize. For example,

```
    (make-instance 'ferror :condition-names '(foo)
                           :format-string "~S loses."
                           :format-args losing-object)
```

creates an instance of **ferror** just like the one that would be signaled if you do

```
    (ferror 'foo "~S loses." losing-object)
```

Note that the flavor name and its components' names are added in automatically to whatever you specify for the **:condition-names** keyword.

Direct use of **make-instance** is cumbersome, however, and it is usually handier to define a *signal name* with **defsignal** or **defsignal-explicit** and then create the instance with **make-condition.**

A signal name is a sort of abbreviation for all the things that are always the same for a certain sort of condition: the flavor to use, the condition names, and what arguments are expected. In addition, it allows you to use a positional syntax for the arguments, which is usually more convenient than a keyword syntax in simple use.

Here is a typical **defsignal**:

```
    (defsignal series-not-convergent sys:arithmetic-error (series)
      "Signaled by limit extractor when SERIES does not converge.")
```

This defines a signal name **series-not-convergent**, together with the name of the flavor to use (**sys:arithmetic-error**, whose meaning is being stretched a little), an interpretation for the arguments (**series**, which is explained below), and a documentation string. The documentation string is not used in printing the error message; it is documentation for the signal name. It becomes accessible via **(documentation 'series-not-convergent 'signal).**

series-not-convergent could then be used to signal an error, or just to create a condition instance:

```
    (ferror 'series-not-convergent
            "The series ~S went to infinity." myseries)

    (make-condition 'series-not-convergent
                    "The series ~S went to infinity." myseries)
```

The list **(series)** in the **defsignal** is a list of implicit instance variable names. They are matched against arguments to **make-condition** following the format string, and each implicit instance variable name becomes an operation defined on the condition instance to return the corresponding argument. (You can imagine that **:gettable-instance-variables** is in effect for all the implicit instance variables.) In this example, sending a **:series** message to the condition instance returns the value specified via **myseries** when the condition was signaled. The implicit instance variables are actually implemented using the condition instance's property list.

Thus, **defsignal** spares you the need to create a new flavor merely in order to remember a particular datum about the condition.

defsignal *Macro*
signal-name (*flavor condition-names...*) *implicit-instance-variables documentation extra-init-keyword-forms*

Defines *signal-name* to create an instance of *flavor* with condition names *condition-names*, and implicit instance variable whose names are taken from the list *implicit-instance-variables* and whose values are taken from the make-condition arguments following the format string.

Instead of a list (*flavor condition-names...*) there may appear just a flavor name. This is equivalent to using *signal-name* as the sole condition name.

The *extra-init-keyword-forms* are forms to be evaluated to produce additional keyword arguments to pass to make-instance. These can be used to initialize other instance variables that particular flavors may have. These expressions can refer to the *implicit-instance-variables*.

documentation is a string which is recorded so that it can be accessed via the function documentation, as in (documentation *signal-name* 'signal).

defsignal-explicit *Macro*
signal-name (*flavor condition-names...*) *signal-arglist documentation init-keyword-forms...*

Like defsignal, defsignal-explicit defines a signal name. This signal name is used the same way, but the way it goes about creating the condition instance is different.

First of all, there is no list of implicit instance variables. Instead, *signal-arglist* is a lambda list which is matched up against all the arguments to make-condition except for the signal-name itself. The variables bound by the lambda list can be used in the *init-keyword-forms*, which are evaluated to get arguments to pass to make-instance. For example:

```
(defsignal-explicit mysignal-3
          (my-error-flavor mysignal-3 my-signals-category)
          (format-string losing-object &rest format-args)
  "The third kind of thing I like to signal."
  :format-string format-string
  :format-args (cons losing-object (copylist format-args))
  :losing-object-name (send losing-object :name))
```

Since implicit instance variables are really just properties on the property list of the instance, you can create them by using init keyword :property-list. The contents of the property list determines what implicit instance variables exist and their values.

make-condition *signal-name* &rest *arguments*

make-condition is the fundamental way that condition instances are created. The *signal-name* says how to interpret the *arguments* and come up with a flavor and values for its instance variables. The handling of the *arguments* is entirely determined by the *signal-name*.

If *signal-name* is a condition instance, make-condition returns it. It is not useful to call make-condition this way explicitly, but this allows condition instances to be passed to the convenience functions error and signal which call make-condition.

If the *signal-name* was defined with defsignal or defsignal-explicit, then that definition specifies exactly how to interpret the *arguments* and create the instance. In general, if the *signal-name* has an eh:make-condition-function property (which is what defsignal defines), this property is a function to which the *signal-name* and *arguments* are passed, and it does the work.

Alternatively, the *signal-name* can be the name of a flavor. Then the *arguments* are passed to make-instance, which interprets them as init keywords and values. This mode is not really recommended and exists for compatibility with Symbolics software.

If the *signal-name* has no eh:make-condition-function property and is not a flavor name, then a trivial defsignal is assumed as a default. It looks like this:

```
(defsignal signal-name ferror ())
```

So the value is an instance of ferror, with the *signal-name* as a condition name, and the *arguments* are interpreted as a format string and args for it.

The *signal-name* nil actually has a definition of this form. nil is frequently used as a signal name in the function ferror when there is no desire to use any condition name in particular.

30.5.3 Signaling a Condition Instance

Once you have a condition instance, you are ready to invoke the condition handling mechanism by signaling it. A condition instance can be signaled any number of times, in any stack groups.

signal-condition *condition-instance* &optional *proceed-types invoke-debugger ucode-error-status inhibit-resume-handlers*

Invoke the condition handling mechanism on *condition-instance*. The list of *proceed-types* says which proceed types (among those conventionally defined for the type of condition you have signaled) you are prepared to implement, should a condition handler return one (see "proceeding"). These are in addition to any proceed types implemented nonlocally by condition-resume forms.

ucode-error-status is used for internal purposes in signaling errors detected by the microcode.

signal-condition tries various possible handlers for the condition. First **eh:condition-handlers** is scanned for handlers that are applicable (according to the condition names they specify) to this condition instance. After this list is exhausted, **eh:condition-default-handlers** is scanned the same way. Each handler that is tried can terminate the act of signaling by throwing out of **signal-condition**, or it can specify a way to proceed from the signal. The handler can also return **nil** to decline to handle the condition; then the next possible handler is offered a chance.

If all handlers decline to handle the condition and *invoke-debugger* is non-**nil**, the debugger is the handler of last resort. With the debugger, the user can ask to throw or to proceed. The default value of *invoke-debugger* is non-**nil** if the *condition-instance* is an error.

If all handlers decline to act and *invoke-debugger* is **nil**, **signal-condition** proceeds using the first proceed type on the list of available ones, provided it is a nonlocal proceed type. If it is a local proceed type, or if there are no proceed types, **signal-condition** just returns **nil**. (It would be slightly simpler to proceed using the first proceed type whether it is local or not. But in the case of a local proceed type, this would just mean returning the proceed type instead of **nil**. It is considered slightly more useful to return **nil**, allowing the signaler to distinguish the case of a condition not handled. The signaler knows which proceed types it specified, and can if it wishes consider **nil** as equivalent to the first of them.)

Otherwise, by this stage, a proceed type has been chosen from the available list. If the proceed type was among those specified by the caller of **signal-condition**, then proceeding consists simply of returning to that caller. The chosen proceed type is the first value, and arguments (returned by the handler along with the proceed type) may follow it. If the proceed type was implemented nonlocally with **condition-resume** (see page 723), then the associated proceed handler function on **eh:condition-resume-handlers** is called.

If *inhibit-resume-handlers* is non-**nil**, resume handlers are not invoked. If a handler returns a nonlocal proceed type, **signal-condition** just returns to its caller as if the proceed type were local. If the condition is not handled, **signal-condition** returns **nil**.

The purpose of **condition-bind-default** is so that you can define a handler that is allowed to handle a signal only if none of the callers' handlers handle it. A more flexible technique for doing this sort of thing is to make an ordinary handler signal the same condition instance recursively by calling **signal-condition**, like this:

```
(multiple-value-list
  (signal-condition condition-instance
                    eh:condition-proceed-types nil nil t))
```

This passes along the same list of proceed types specified by the original signaler, prevents the debugger from being called, and prevents resume handlers from being run. If the first value **signal-condition** returns is non-**nil**, one of the outer handlers has handled the condition; your handler's simplest option is to return those same values so that the other handler has its way (but it could also examine them and return modified values). Otherwise, you go on to handle the condition in your default manner.

eh:trace-conditions *Variable*

This variable may be set to a list of condition names to be *traced*. Whenever a condition possessing a traced condition name is signaled, an error is signaled to report the fact. (Tracing of conditions is turned off while this error is signaled and handled). Proceeding with proceed type :no-action causes the signaling of the original condition to continue.

If eh:trace-conditions is t, all conditions are traced.

30.6 Proceeding

Both condition handlers and the user (through the debugger) have the option of proceeding certain conditions.

Each condition name can define, as a convention, certain *proceed types*, which are keywords that signify a certain conceptual way to proceed. For example, condition name sys:wrong-type-argument defines the proceed type :argument-value which means, "Here is a new value to use as the argument."

Each signaler may or may not implement all the proceed types which are meaningful in general for the condition names being signaled. For example, it is futile to proceed from a sys:wrong-type-argument error with :argument-value unless the signaler knows how to take the associated value and store it into the argument, or do something else that fits the conceptual specifications of :argument-value. For some signalers, it may not make sense to do this at all. Therefore, one of the arguments to signal-condition is a list of the proceed types that this particular signaler knows how to handle.

In addition to the proceed types specified by the individual signaler, other proceed types can be provided nonlocally; they are implemented by a *resume handler* which is in effect through a dynamic scope. See below, section 30.6.3, page 723.

A condition handler can use the operations :proceed-types and :proceed-type-p on the condition instance to find out which proceed types are available. It can request to proceed by returning one of the available proceed types as a value. This value is returned from signal-condition, and the condition's signaler can take action as appropriate.

If the handler returns more than one value, the remaining values are considered *arguments* of the proceed type. The meaning of the arguments to a proceed type, and what sort of arguments are expected, are part of the conventions associated with the condition name that gives the proceed type its meaning. For example, the :argument-value proceed type for sys:wrong-type-argument errors conventionally takes one argument, which is the new value to use. All the values returned by the handler are returned by signal-condition to the signaler.

Here is an example of a condition handler that proceeds from sys:wrong-type-argument errors. It makes any atom effectively equivalent to nil when used in car or any other function that expects a list. The handler uses the :description operation, which on sys:wrong-type-argument condition instances returns a keyword describing the data type that was desired.

```
(condition-bind
    ((sys:wrong-type-argument
         #'(lambda (condition)
                (if (eq (send condition :description) 'cons)
                    (values :argument-value nil)))))
  body...)
```

Here the argument to the :argument-value proceed type is nil.

:proceed-types *Operation on* condition

Returns a list of the proceed types available for this condition instance. This operation should be used only within the signaling of the condition instance, as it refers to the special variable in which signal-condition stores its second argument.

:proceed-type-p *proceed-type* *Operation on* condition

t if *proceed-type* is one of the proceed types available for this condition instance. This operation should be used only within the signaling of the condition instance, as it refers to the special variable in which signal-condition stores its second argument.

30.6.1 Proceeding and the Debugger

If the condition invokes the debugger, then the user has the opportunity to proceed. When the debugger is entered, the available proceed types are assigned command characters starting with Super-A. Each character becomes a command to proceed using the corresponding proceed type.

Three additional facilities are required to make it convenient for the user to proceed using the debugger. Each is provided by methods defined on condition flavors. When you define a new condition flavor, you must provide methods to implement these facilities.

Documentation:

It must be possible to tell the user what each proceed type is *for*.

Prompting for arguments:

The user must be asked for the arguments for the proceed type. Each proceed type may have different arguments to ask for.

Not always the same proceed types:

Usually the user can choose among the same set of proceed types that a handler can, but sometimes it is useful to provide the user with a few extra ones, or to suppress some of them for him.

These three facilities are provided by methods defined on condition flavors. Each proceed type that is provided by signalers should be accompanied by suitable methods. This means that you must normally define a new flavor if you wish to use a new proceed type.

The :document-proceed-type operation is supposed to print documentation of what a proceed type is for. For example, when sent to a condition instance describing an unbound variable error, if the proceed type specified is :new-value, the text printed is something like "Proceed, reading a value to use instead."

:document-proceed-type *proceed-type stream* *Operation on* condition

Prints on *stream* a description of the purpose of proceed type *proceed-type*. This operation uses :case method combination (see section 21.11, page 433), to make it convenient to define the way to document an individual proceed type. The string printed should start with an imperative verb form, capitalized, and end with a period. Example:

This example is an :or method so that it can consider any proceed type. If it returns non-nil, the system considers that it has handled the proceed type and no other methods get a chance. eh:places is an instance variable of the flavor sys:failed-assertion; its values are the proceed types this method understands.

```
(defmethod (sys:failed-assertion :or :document-proceed-type)
           (proceed-type stream ignore)
  (when (memq proceed-type eh:places)
    (format stream
            "Try again, setting ~S.~
             You type an expression for it."
            proceed-type)
    t))
```

As a last resort, if the condition instance has a :case method for :proceed-asking-user with *proceed-type* as the suboperation, and this method has a documentation string, it is printed. This is in fact the usual way that a proceed type is documented.

The :proceed-asking-user operation is supposed to ask for suitable arguments to pass with the proceed type. Sending :proceed-asking-user to an instance of sys:unbound-variable with argument :new-value would read and evaluate one expression, prompting appropriately.

:proceed-asking-user *Operation on* condition
proceed-type continuation read-object-fn

The method for :proceed-asking-user embodies the knowledge of how to prompt for and read the additional arguments that go with *proceed-type*.

:case method combination is used (see section 21.11, page 433), making it possible to define the handling of each proceed type individually in a separate function. The documentation string of the :case method for a proceed type is also used as the default for :document-proceed-type on that proceed type.

The argument *continuation* is an internal function of the debugger which actually proceeds from the signaled condition if the :proceed-asking-user method calls it. This is the only way to cause proceeding actually to happen. Call *continuation* with funcall, giving a proceed type and suitable arguments. The proceed type passed to *continuation* need not be the same as the one given to :proceed-asking-user; it should be one of the proceed types available for handlers to use.

Alternatively, the :prompt-and-read method can return without calling *continuation*; then the debugger continues to read commands. The options which the fs:no-more-room condition offers in the debugger, to run Dired or expunge a directory, work this way.

The argument *read-object-fn* is another internal function of the debugger whose purpose is to read arguments from the user or request confirmation. If you wish to do those things, you must **funcall** *read-object-function* to do it. Use the calling sequence documented for the function **prompt-and-read** (see page 453). (The *read-object-fn* may or may not actually use **prompt-and-read**.)

Here is how **sys:proceed-with-value-mixin** provides for the proceed type **:new-value**. Note the documentation string, which is automatically use by **:document-proceed-type** since no **:case** method for that operation is provided.

```
(defmethod (proceed-with-value-mixin
                   :case :proceed-asking-user :new-value)
           (continuation read-object-function)
  "Return a value; the value of an expression you type."
  (funcall continuation :new-value
           (funcall read-object-function
                    :eval-read
                    "~&Form whose value to use instead: ")))
```

The **:user-proceed-types** operation is given the list of proceed types actually available and is supposed to return the list of proceed types to offer to the user. By default, this operation returns its argument: all proceed types are available to the user through the debugger.

For example, the condition name **sys:unbound-variable** conventionally defines the proceed types **:new-value** and **:no-action**. The first specifies a new value; the second attempts to use the variable's current value and gets another error if the variable is still unbound. These are clean operations for handlers to use. But it is more convenient for the user to be offered only one choice, which will use the variable's new value if it is bound now, but otherwise ask for a new value. This is implemented with a **:user-proceed-types** method that replaces the two proceed types with a single one.

Or, you might wish to offer the user two different proceed types that differ only in how they ask the user for additional information. For handlers, there would be only one proceed type.

Finally, there may be proceed types intended only for the debugger which do not actually proceed; these should be inserted into the list by the **:user-proceed-types** method.

:user-proceed-types *proceed-types* *Operation on* **condition**

Assuming that *proceed-types* is the list of proceed types available for condition handlers to return, this operation returns the list of proceed types that the debugger should offer to the user.

Only the proceed types offered to the user need to be handled by **:document-proceed-type** and **:proceed-asking-user**.

The flavor **condition** itself defines this to return its argument. Other condition flavors may redefine this to filter the argument in some appropriate fashion.

:pass-on method combination is used (see section 21.11, page 433), so that if multiple mixins define methods for :user-proceed-types, each method gets a chance to add or remove proceed types. The methods should not actually modify the argument, but should cons up a new list in which certain keywords are added or removed according to the other keywords that are there.

Elements should be removed only if they are specifically recognized. This is to say, the method should make sure that any unfamiliar elements present in the argument are also present in the value. Arranging to omit certain specific proceed types is legitimate; returning only the intersection with a constant list is not legitimate.

Here is an example of nontrivial use of :user-proceed-types:

```
(defflavor my-error () (error))

(defmethod (my-error :user-proceed-types) (proceed-types)
  (if (memq :foo proceed-types)
      (cons :foo-two-args proceed-types)
    proceed-types))

(defmethod (my-error :case :proceed-asking-user :foo)
           (cont read-object-fn)
  "Proceeds, reading a value to foo with."
  (funcall cont :foo
           (funcall read-object-fn :eval-read
                    "Value to foo with: ")))

(defmethod (my-error :case :proceed-asking-user :foo-two-args)
           (cont read-object-fn)
  "Proceeds, reading two values to foo with."
  (funcall cont :foo
           (funcall read-object-fn :eval-read
                    "Value to foo with: ")
           (funcall read-object-fn :eval-read
                    "Value to foo some more with: ")))
```

In this example, if the signaler provides the proceed type :foo, then it is described for the user as "proceeds, reading a value to foo with"; and if the user specifies that proceed type, he is asked for a single value, which is used as the argument when proceeding. In addition, the user is offered the proceed type :foo-two-args, which has its own documentation and which reads two values. But for condition handlers there is really only one proceed type, :foo. :foo-two-args is just an alternate interface for the user to proceed type :foo, and this is why the :user-proceed-types method offers :foo-two-args only if the signaler is willing to accept :foo.

30.6.2 How Signalers Provide Proceed Types

Each condition name defines a conceptual meaning for certain proceed types, but this does not mean that all of those proceed types may be used every time the condition is signaled. The signaler must specifically implement the proceed types in order to make them do what they are conventionally supposed to do. For some signalers it may be difficult to do, or may not even make sense. For example, it is no use having a proceed type **:store-new-value** if the signaler does not have a suitable place to store, permanently, the argument the handler supplies.

Therefore, we require each signaler to specify just which proceed types it implements. Unless the signaler explicitly specifies proceed types one way or another, no proceed types are allowed (except for nonlocal ones, described in the following section).

One way to specify the proceed types allowed is to call **signal-condition** and pass the list of proceed types as the second argument.

Another way that is less general but more convenient is **signal-proceed-case**.

signal-proceed-case ((*variables...*) *make-condition-arguments...*) *clauses...* *Macro*

Signals a condition, providing proceed types and code to implement them. Each clause specifies a proceed type to provide, and also contains code to be run if a handler should proceed with that proceed type.

```
        (signal-proceed-case ((argument-vars...)
                              signal-name signal-name-arguments...)
          (proceed-type forms...)
          (proceed-type forms...)
          ...)
```

A condition-object is created with **make-condition** using the *signal-name* and *signal-name-arguments*; then it is signaled giving a list of the proceed types from all the clauses as the list of proceed types allowed.

The variables *argument-vars* are bound to the values returned by **signal-condition**, except for the first value, which is tested against the *proceed-type* from each clause, using a **selectq**. The clause that matches is executed.

Example:

```
(defsignal my-wrong-type-arg
           (eh:wrong-type-argument-error sys:wrong-type-argument)
  (old-value arg-name description)
  "Wrong type argument from my own code.")

(signal-proceed-case
        ((newarg)
         'my-wrong-type-arg
         "The argument ~A was ~S, which is not a cons."
         'foo foo 'cons)
  (:argument-value (car newarg)))
```

The signal name my-wrong-type-arg creates errors with condition name sys:wrong-type-argument. The signal-proceed-case shown signals such an error, and handles the proceed type :argument-value. If a handler proceeds using that proceed type, the handler's value is put in newarg, and then its car is returned from the signal-proceed-case.

30.6.3 Nonlocal Proceed Types

When the caller of signal-condition specifies proceed types, these are called *local proceed types*. They are implemented at the point of signaling. There are also *nonlocal proceed types*, which are in effect for all conditions (with appropriate condition names) signaled during the execution of the body of the establishing macro. We say that the macro establishes a *resume handler* for the proceed type.

The most general construct for establishing a resume handler is condition-resume. For example, in

```
(condition-resume
    '(fs:file-error :retry-open t
                    ("Proceeds, opening the file again.")
                    (lambda (ignore) (throw 'tag nil)))
  (do-forever
    (catch 'tag (return (open pathname)))))
```

the proceed type :retry-open is available for all fs:file-error conditions signaled within the call to open.

condition-resume *handler-form* &body *body* *Macro*
condition-resume-if *cond-form* *handler-form* &body *body* *Macro*

Both execute *body* with a resume handler in effect for a nonlocal proceed type according to the value of *handler-form*. For condition-resume-if, the resume handler is in effect only if *cond-form*'s value is non-nil.

The value of the *handler-form* should be a list with at least five elements:

(*condition-names proceed-type predicate format-string-and-args handler-function additional-args*. . .)

condition-names is a condition name or a list of them. The resume handler applies to these conditions only.

proceed-type is the proceed type implemented by this resume handler.

predicate is either t or a function that is applied to a condition instance and determines whether the resume handler is in effect for that condition instance.

format-string-and-args is a list of a string and additional arguments that can be passed to format to print a description of what this proceed type is for. These are needed only for anonymous proceed types.

handler-function is the function called to do the work of proceeding, once this proceed type has been returned by a condition handler or the debugger.

catch-error-restart-explicit-if makes it easy to establish a particular simple kind of resume handler.

catch-error-restart-explicit-if *Macro*
cond-form (*condition-names proceed-type format-string format-args...*) *body...*

Executes *body* with (if *cond-form* produces a non-nil value) a resume handler for proceed type *proceed-type* and condition(s) *condition-names*. *condition-names* should be a symbol or a list of symbols; it is not evaluated. *proceed-type* should be a symbol.

If proceeding is done using this resume handler, control returns from the catch-error-restart-explicit-if form. The first value is nil and the second is non-nil.

format-string and the *format-args*, all of which are evaluated, are used by the :document-proceed-type operation to describe the proceed type, if it is anonymous.

For condition handlers there is no distinction between local and nonlocal proceed types. They are both included in the list of available proceed types returned by the :proceed-types operation (all the local proceed types come first), and the condition handler selects one by returning the proceed type and any conventionally associated arguments. The debugger's :user-proceed-types, :document-proceed-type and :proceed-asking-user operations are also make no distinction.

The difference comes after the handler or the debugger returns to signal-condition. If the proceed type is a local one (one of those in the second argument to signal-condition), signal-condition simply returns. If the proceed type is not among those the caller handles, signal-condition looks for a resume handler associated with the proceed type, and calls its handler function. The arguments to the handler function are the condition instance, the *additional-args* specified in the resume handler, and any arguments returned by the condition handler in addition to the proceed type. The handler function is supposed to do a throw. If it returns to signal-condition, an error is signaled.

You are allowed to use "anonymous" nonlocal proceed types, which have no conventional meaning and are not specially known to the :document-proceed-type and :proceed-asking-user operations. The anonymous proceed type may be any Lisp object. The default definition of :proceed-asking-user handles an anonymous proceed type by simply calling the continuation passed to it, reading no arguments. The default definition of :document-proceed-type handles anonymous proceed types by passing format the *format-string-and-args* list found in the resume handler (this is what that list is for).

Anonymous proceed types are often lists. Such proceed types are usually made by some variant of error-restart, and they are treated a little bit specially. For one thing, they are all put at the end of the list returned by the :proceed-types operation. For another, the debugger command Control-C or Resume never uses a proceed type which is a list. If no atomic proceed type is available, Resume or Control-C is not allowed.

error-restart (*condition-names format-string format-args...*) *body...* *Macro*
error-restart-loop *Macro*
catch-error-restart *Macro*
catch-error-restart-if *Macro*
cond-form (*condition-names format-string format-args...*) *body...*

All execute body with an anonymous resume handler for *condition-names*. The proceed type for this resume handler is a list, so the **Resume** key will not use it. *condition-names* is either a single condition name or a list of them, or **nil** meaning all conditions; it is not evaluated.

format-string and the *format-args*, all of which are evaluated, are used by the **:document-proceed-type** operation to describe the anonymous proceed type.

If the resume handler made by **error-restart** is invoked by proceeding from a signal, the automatically generated resume handler function does a throw back to the **error-restart** and the body is executed again from the beginning. If body returns, the values of the last form in it are returned from the **error-restart** form.

error-restart-loop is like **error-restart** except that it loops to the beginning of body even if body completes normally. It is like enclosing an **error-restart** in an iteration.

catch-error-restart is like **error-restart** except that it never loops back to the beginning. If the anonymous proceed type is used for proceeding, the **catch-error-restart** form returns with **nil** as the first value and a non-**nil** second value.

catch-error-restart-if is like **catch-error-restart** except that the resume handler is only in effect if the value of the *cond-form* is non-**nil**.

All of these variants of **error-restart** can be written in terms of **condition-resume-if.**

These forms are typically used by any sort of command loop, so that aborting within the command loop returns to it and reads another command. **error-restart-loop** is often right for simple command loops. **catch-error-restart** is useful when aborting should terminate execution rather than retry, or with an explicit conditional to test whether a throw was done.

error-restart forms often specify **(error sys:abort)** as the *condition-names*. The presence of **error** causes them to be listed (and assigned command characters) by the debugger, for all errors, and the presence of **sys:abort** causes the **Abort** key to use them. If you would like a procede type to be offered as an option by the debugger, but do not want the **Abort** key to use it, specify just **error.**

eh:invoke-resume-handler *condition-instance proceed-type* &rest *args*

Invokes the innermost applicable resume handler for *proceed-type*. Applicability of a resume handler is determined by matching its condition names against those possessed by *condition-instance* and by applying its predicate, if not **t**, to *condition-instance*.

If *proceed-type* is **nil**, the innermost applicable resume handler is invoked regardless of its proceed type. However, in this case, the scan stops if **t** is encountered as an element of **eh:condition-resume-handlers.**

eh:condition-resume-handlers *Variable*

The current list of resume handlers for nonlocal proceed types. **condition-resume** works by binding this variable. Elements are usually lists that have the format described above under **condition-resume**. The symbol **t** is also meaningful as an element of this list. It terminates the scan for a resume handler when it is made by **signal-condition** for a condition that was not handled. **t** is pushed onto the list by break loops and the debugger to shield the evaluation of your type-in from automatic invocation of resume handlers established outside the break loop or the error.

The links of this list, and its elements, are often created using **with-stack-list**. so be careful if you try to save the value outside the context in which you examine it.

sys:abort (condition) *Condition*

This condition is signaled by the **Abort** key; it is how that key is implemented. Most command loops use some version of **error-restart** to set up a resume handler for **sys:abort** so that it will return to the innermost command loop if (as is usually the case) no handler handles it. These resume handlers usually apply to **error** as well as **sys:abort**, so that the debugger will offer a specific command to return to the command loop even if it is not the innermost one.

30.7 The Debugger

When an error condition is signaled and no handlers decide to handle the error, an interactive debugger is entered to allow the user to look around and see what went wrong, and to help him continue the program or abort it. This section describes how to use the debugger.

30.7.1 Entering the Debugger

There are two kinds of errors; those generated by the Lisp Machine's microcode, and those generated by Lisp programs (by using **ferror** or related functions). When there is a microcode error, the debugger prints out a message such as the following:

```
>>TRAP 5543 (TRANS-TRAP)
The symbol FOOBAR is unbound.
While in the function LOSE-XCT ← LOSE-COMMAND-LOOP ← LOSE
```

The first line of this error message indicates entry to the debugger and contains some mysterious internal microcode information: the micro program address, the microcode trap name and parameters, and a microcode backtrace. Users can ignore this line in most cases. The second line contains a description of the error in English. The third line indicates where the error happened by printing a very abbreviated "backtrace" of the stack (see below); in the example, it is saying that the error was signaled inside the function **lose-xct**, which was called by **lose-command-loop**.

Here is an example of an error from Lisp code:

```
>>ERROR: The argument X was 1, which is not a symbol,
While in the function FOO ← SI:EVAL1 ← SI:LISP-TOP-LEVEL1
```

Here the first line contains the English description of the error message, and the second line contains the abbreviated backtrace. **foo** signaled the error by calling **ferror**; however, **ferror** is censored out of the backtrace.

After the debugger's initial message, it prints the function that got the error and its arguments. Then it prints a list of commands you can use to proceed from the error, or to abort to various command loops. The possibilities depend on the kind of error and where it happened, so the list is different each time; that is why the debugger prints it. The commands in the list all start with **Super-A**, **Super-B** and continue as far as is necessary.

eh:*inhibit-debugger-proceed-prompt* *Variable*

If this is non-**nil**, the list of **Super** commands is not printed when the debugger is entered. Type **Help P** to see the list.

The debugger normally uses the stream ***debug-io*** for all its input and output (see page 460). By default it is a synonym for ***terminal-io***. The value of this variable in the stack group in which the error was signaled is the one that counts.

eh:*debug-io-override* *Variable*

If this is non-**nil**, it is used by the debugger instead of the value of ***debug-io***. The value in the stack group where the error was signaled is the one that counts.

The debugger can be manually entered either by causing an error (e.g. by typing a ridiculous symbol name such as **ahsdgf** at the Lisp read-eval-print loop) or by typing the **Break** key with the **Meta** shift held down while the program is reading from the terminal. Typing the **Break** key with both **Control** and **Meta** held down forces the program into the debugger immediately, even if it is running. If the **Break** key is typed without **Meta**, it puts you into a read-eval-print loop using the **break** function (see page 795) rather than into the debugger.

eh &optional *process*

Causes *process* to enter the debugger, and directs the debugger to read its commands from the ambient value of ***terminal-io***, current when you call **eh**, rather than *process* 's own value of ***terminal-io*** which is what would be used if *process* got an error in the usual way. The process in which you invoked **eh** waits while you are in the debugger, so there is no ambiguity about which process will handle your keyboard input.

If *process* had already signaled an error and was waiting for exposure of a window, then it enters the debugger to handle that error. Otherwise, the **break** condition is signaled in it (just like what **Control-Meta-Break** does) to force it into the debugger.

The **Resume** command makes *process* resume execution. You can also use other debugger commands such as **Abort**, **Control-R**, **Control-Meta-R** and **Control-T** to start it up again. Exiting the debugger in any way causes **eh** to return in its process.

process can also be a window, or any flavor instance which understands the **:process** operation and returns a process.

If *process* is not a process but a stack group, the current state of the stack group is examined. In this case, the debugger runs in "examine-only" mode, and executes in the process in which you invoked **eh**. You cannot resume execution of the debugged stack group, but **Resume** exits the debugger. It is your responsibility to ensure that no one tries to execute in the stack group being debugged while the debugger is looking at it.

If *process* is **nil**, **eh** finds all the processes waiting to enter the debugger and asks you which one to use.

30.7.2 How to Use the Debugger

Once inside the debugger, the user may give a wide variety of commands. This section describes how to give the commands, then explains them in approximate order of usefulness. A summary is provided at the end of the listing.

When the debugger is waiting for a command, it prompts with an arrow:

```
→
```

If the error took place in the evaluation of an expression that you typed at the debugger, you are in a second level (or deeper) error. The number of arrows in the prompt indicates the depth.

The debugger also warns you about certain unusual circumstances that may cause paradoxical results. For example, if **default-cons-area** is anything except **working-storage-area**, a message to that effect is printed. If ***read-base*** and ***print-base*** are not the same, a message is printed.

At this point, you may type either a Lisp expression or a command; a **Control** or **Meta** character is interpreted as a command, whereas most normal characters are interpreted as the first character of an expression. If you type the **Help** key or the **?** key, you can get some introductory help with the debugger.

If you type a Lisp expression, it is interpreted as a Lisp form and evaluated in the context of the current frame. That is, all dynamic bindings used for the evaluation are those in effect in the current frame, with certain exceptions explained below. The results of the evaluation are printed, and the debugger prompts again with an arrow. If, during the typing of the form, you change your mind and want to get back to the debugger's command level, type the **Abort** key or a **Control-G**; the debugger responds with an arrow prompt. In fact, at any time that input is expected from you, while you are in the debugger, you may type **Abort** or **Control-G** to cancel any debugger command that is in progress and get back to command level. **Control-G** is useful because it can never exit from the debugger as **Abort** can.

This **read-eval-print** loop maintains the values of **+**, *****, and **-** almost like the Lisp listen loop. The difference is that some single-character debugger commands such as **C-M-A** also set ***** and **+** in their own way.

If an error occurs in the evaluation of the Lisp expression you type, you may enter a second invocation of the debugger, looking at the new error. The prompt in this event is '→→' to make it clear which level of error you are examining. You can abort the computation and get back to the first debugger level by typing the **Abort** key (see below).

Various debugger commands ask for Lisp objects, such as an object to return or the name of a catch-tag. You must type a form to be evaluated; its value is the object that is actually used. This provides greater generality, since there are objects to which you might want to refer that cannot be typed in (such as arrays). If the form you type is non-trivial (not just a constant form), the debugger shows you the result of the evaluation and asks for confirmation before proceeding. If you answer negatively, or if you abort, the command is canceled and the debugger returns to command level. Once again, the special bindings in effect for evaluation of the form are those of the current frame you have selected.

The **Meta-S** and **Control-Meta-S** commands allow you to look at the bindings in effect at the current frame. A few variables are rebound by the debugger itself whenever a user-provided form is evaluated, so you you must use **Meta-S** to find the values they actually had in the erring computation.

terminal-io ***terminal-io*** is rebound to the stream the debugger is using.

standard-input
standard-output

standard-input and ***standard-output*** are rebound to be synonymous with ***terminal-io***.

+, ++, +++
, **, ***, *values

+ and ***** are rebound to the debugger's previous form and previous value. Commands for examining arguments and such, including **C-M-A**, **C-M-L** and **C-M-V**, leave ***** set to the value examined and **+** set to a locative to where the value was found. When the debugger is first entered, **+** is the last form typed, which is typically the one that caused the error, and ***** is the value of the *previous* form.

evalhook
applyhook These variables (see page 748) are rebound to **nil**, turning off the **step** facility if it was in use when the error occurred.

eh:condition-handlers
eh:condition-default-handlers

These are rebound to **nil**, so that errors occurring within forms you type while in the debugger do not magically resume execution of the erring program.

eh:condition-resume-handlers

To prevent resume handlers established outside the error from being invoked automatically by deeper levels of error, this variable is rebound to a new value, which has an element **t** added in the front.

30.7.3 Debugger Commands

All debugger commands are single characters, usually with the **Control** or **Meta** bits. The single most useful command is **Abort** (or **Control-Z**), which exits from the debugger and throws out of the computation that got the error. (This is the **Abort** key, not a 5-letter command.) Often you are not interested in using the debugger at all and just want to get back to Lisp top level; so you can do this in one keystroke.

If the error happened while you were innocently using a system utility such as the editor, then it represents a bug in the system. Report the bug using the debugger command **Control-M**. This gives you an editor preinitialized with the error message and a backtrace. You should type in a *precise* description of what you did that led up to the problem, then send the message by typing **End**. Be as complete as possible, and always give the exact commands you typed, exact filenames, etc. rather then general descriptions, as much as possible. The person who investigates the bug report will have to try to make the problem happen again; if he does not know where to find *your* file, he will have a difficult time.

The **Abort** command signals the **sys:abort** condition, returning control to the most recent command loop. This can be Lisp top level, a **break**, or the debugger command loop associated with another error. Typing **Abort** multiple times throws back to successively older read-eval-print or command loops until top level is reached. Typing **Meta-Abort**, on the other hand, always throws to top level. **Meta-Abort** is not a debugger command, but a system command that is always available no matter what program you are in.

Note that typing **Abort** in the middle of typing a form to be evaluated by the debugger aborts that form and returns to the debugger's command level, while typing **Abort** as a debugger command returns out of the debugger and the erring program, to the *previous* command level. Typing **Abort** after entering a numeric argument just discards the argument.

Self-documentation is provided by the **Help** or **?** command, which types out some documentation on the debugger commands, including any special commands that apply to the particular error currently being handled.

Often you want to try to proceed from the error. When the debugger is entered, it prints a table of commands you can use to proceed, or abort to various levels. The commands are **Super-A**, **Super-B**, and so on. How many there are and what they do is different each time there is an error, but the table says what each one is for. If you want to see the table again, type **Help** followed by **P**.

The **Resume** (or **Control-C**) command is often synonymous with **Super-A**. But **Resume** only proceeds, never aborts. If there is no way to proceed, just ways to abort, then **Resume** does not do anything.

The debugger knows about a current stack frame, and there are several commands that use it. The initially current stack frame is the one which signaled the error, either the one which got the microcode-detected error or the one which called **ferror**, **cerror**, or **error**. When the debugger starts it up it shows you this frame in the following format:

```
FOO:
   Arg 0 (X): 13
   Arg 1 (Y): 1
```

and so on. This means that foo was called with two arguments, whose names (in the Lisp source code) are x and y. The current values of x and y are 13 and 1 respectively. These may not be the original arguments if foo happens to setq its argument variables.

The Clear-Screen (or Control-L) command clears the screen, retypes the error message that was initially printed when the debugger was entered, and prints out a description of the current frame, in the above format.

Several commands are provided to allow you to examine the Lisp control stack and to make frames current other than the one that got the error. The control stack (or "regular pdl") keeps a record of all functions currently active. If you call foo at Lisp's top level, and it calls bar, which in turn calls baz, and baz gets an error, then a backtrace (a backwards trace of the stack) would show all of this information. The debugger has two backtrace commands. Control-B simply prints out the names of the functions on the stack; in the above example it would print

```
BAZ ← BAR ← FOO ← SI:*EVAL
 ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

The arrows indicate the direction of calling. The Meta-B command prints a more extensive backtrace, indicating the names of the arguments to the functions and their current values; for the example above it might look like:

```
BAZ:
   Arg 0 (X): 13
   Arg 1 (Y): 1

BAR:
   Arg 0 (ADDEND): 13

FOO:
   Arg 0 (FROB): (A B C . D)
```

and so on. The backtrace commands all accept numeric arguments which say how many frames to describe, the default being to describe all the frames.

Moving around in the stack:

The Control-N command makes the "next" older frame be current. This is the frame which called the one that was current at before. The new current frame's function and arguments are printed in the format shown immediately above.

Control-P moves the current frame in the reverse direction. If you use it immediately after getting an error it selects frames that are part of the act of signaling.

Meta-< selects the frame in which the error occurred, the same frame that was selected when the debugger was entered. Meta-> selects the outermost or initial stack frame. Control-S asks you for a string and searches down the stack (toward older frames) from the current frame for a frame whose executing function's name contains that string. That frame becomes current and is printed out. These commands are easy to remember since they are analogous to editor commands.

The Control-Meta-N, Control-Meta-P, and Control-Meta-B commands are like the corresponding Control commands but don't censor the stack to omit "uninteresting" functions. When looking at interpreted code, the debugger usually tries to skip over frames that belong to the functions composing the interpreter, such as eval, prog, and cond. Control-Meta-N, Control-Meta-P, and Control-Meta-B show everything. They also show frames that are not yet active; that is, frames whose arguments are still being computed for functions that are going to be called. The Control-Meta-U command goes down the stack (to older frames) to the next interesting function and makes that the current frame.

Meta-L prints out the current frame in "full screen" format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. Refer to chapter 31, page 752 for help in reading this machine code.

Commands such as Control-N and Control-P, which are useful to issue repeatedly, take a prefix numeric argument and repeat that many types. The numeric argument is typed by using Control or Meta and the number keys, as in the editor. Some other commands such as Control-M also use the numeric argument; refer to the table at the end of the section for detailed information.

Resuming execution:

Meta-C is similar to Control-C, but in the case of an unbound variable or undefined function, actually setqs the variable or defines the function, so that the error will not happen again. Control-C (or Resume) provides a replacement value but does not actually change the variable. Meta-C proceeds using the proceed type :store-new-value, and is available only if that proceed type is provided.

Control-R is used to return a value or values from the current frame; the frame that called that frame continues running as if the function of the current frame had returned. This command prompts you for each value that the caller expects; you can type either a form which evaluates to the desired value or End if you wish to return no more values.

The Control-T command does a throw to a given tag with a given value; you are prompted for the tag and the value.

Control-Meta-R *reinvokes* the current frame; it starts execution at the beginning of the function, with the arguments currently present in the stack frame. These are the same arguments the function was originally called with unless either the function itself has changed them with setq or you have set them in the debugger. If the function has been redefined in the meantime (perhaps you edited it and fixed its bug) the new definition is used. Control-Meta-R asks for confirmation before resuming execution.

Meta-R is similar to Control-Meta-R but allows you to change the arguments if you wish. You are prompted for the new arguments one by one; you can type a form which evaluates to the desired argument, or Space to leave that argument unchanged, or End if you do not want any more arguments. Space is allowed only if this argument was previously passed, and End is not allowed for a required argument. Once you have finished specifying the arguments, you must confirm before execution resumes.

Stepping through function calls and returns:

You can request a trap to the debugger on exit from a particular frame, or the next time a function is called.

Each stack frame has a "trap on exit" bit. The Control-X command toggles this bit. The Meta-X command sets the bit to cause a trap for the current frame and all outer frames. If a program is in an infinite loop, this is a good way to find out how far back on the stack the loop is taking place. This also enables you to see what values are being returned. The Control-Meta-X command clears the trap-on-exit bit for the current frame and outer frames.

The Control-D command proceeds like Control-C but requests a trap the next time a function is called. The Meta-D command toggles the trap-on-next-call bit for the erring stack group. It is useful if you wish to set the bit and then resume execution with something other than Control-C. The function breakon may be used to request a trap on calling a particular function. Trapping on entry to a frame automatically sets the trap-on-exit bit for that frame; use Control-X to clear it if you do not want another trap.

Transfering to other systems:

Control-E puts you into the editor, looking at the source code for the function in the current frame. This is useful when you have found the function that caused the error and that needs to be fixed. The editor command Control-Z will return to the debugger, if it is still there.

Control-M puts you into the editor to mail a bug report. The error message and a backtrace are put into the editor buffer for you. A numeric argument says how many frames to include in the backtrace.

Control-Meta-W calls the window debugger, a display-oriented debugger. It is not documented in this manual, but should be usable without further documentation.

Examining and setting the arguments, local variables, and values:

Control-Meta-A takes a numeric argument, *n*, and prints out the value of the *n*th argument of the current frame. It leaves * set to the value of the argument, so that you can use the Lisp read-eval-print loop to examine it. It also leaves + set to a locative pointing to the argument on the stack, so that you can change that argument (by calling rplacd on the locative). Control-Meta-L is similar, but refers to the *n*th local variable of the frame. Control-Meta-V refers to the *n*th value this frame has returned (in a trap-on-exit). Control-Meta-F refers to the function executing in the frame; it ignores its numeric argument and doesn't allow you to change the function.

Another way to examine and set the arguments, locals and values of a frame is with the functions eh-arg, eh-loc, eh-val and eh-fun. Use these functions in expressions you evaluate inside the debugger, and they refer to the arguments, locals, values and function, respectively, of the debugger's current frame.

The names eh:arg, eh:val, etc. are for compatibility with the Symbolics system.

eh-arg *arg-number-or-name*
eh:arg *arg-number-or-name*

When used in an expression evaluated in the debugger, **eh-arg** returns the value of the specifed argument in the debugger's current frame. Argument names are compared ignoring packages; only the pname of the symbol you supply is relevant. **eh-arg** can appear in **setf** and **locf** to set an argument or get its location.

eh-loc *local-number-or-name*
eh:loc *local-number-or-name*

Like **eh-arg** but accesses the current frame's local variables instead of its arguments.

eh-val &optional *value-number-or-name*
eh:val &optional *value-number-or-name*

eh-val is used in an expression evaluated in the debugger when the current frame is returning multiple values, to examine those values. This is only useful if the function has already begun to return some values (as in a trap-on-exit), since otherwise they are all **nil**. If a name is specified, it is looked for in the function's **values** or **return-list** declaration, if any.

eh-val can be used with **setf** and **locf**. You can make a frame return a specific sequence of values by setting all but the last value with **eh-val** and doing Control-R to return the last value.

eh-val with no argument returns a list of all the values this frame is returning.

eh-fun
eh:fun

eh-fun can be called in an expression being evalued inside the debugger to return the function-object being called in the current frame. It can be used with **setf** and **locf**.

30.7.4 Summary of Commands

Control-A	Prints argument list of function in current frame.
Control-Meta-A	Sets * to the *n*th argument of the current frame.
Control-B	Prints brief backtrace.
Meta-B	Prints longer backtrace.
Control-Meta-B	Prints longer backtrace with no censoring of "uninteresting" functions.
Control-C or Resume	Attempts to continue, using the first proceed type on the list of available ones for this error.
Meta-C	Attempts to continue, setq'ing the unbound variable or otherwise "permanently" fixing the error. This uses the proceed type :store-new-value, and is available only if that proceed type is.

Control-D	Attempts to continue like Control-C, but trap on the next function call.
Meta-D	Toggles the flag that causes a trap on the next function call after you continue or otherwise exit the debugger.
Control-E	Switches to Zmacs to edit the source code for the function in the current frame.
Control-Meta-F	Sets * to the function in the current frame.
Control-G or Abort	Quits to command level. This is not a command, but something you can type to escape from typing in an argument of a command.
Control-Meta-H	Describes the condition handlers and resume handlers established by the current frame.
Control-L or Clear-Screen	Redisplays error message and current frame.
Meta-L	Displays the current frame, including local variables and compiled code.
Control-Meta-L	Sets * to the value of local variable *n* of the current frame.
Control-M	Sends a bug report containing the error message and a backtrace of *n* frames (default is 3).
Control-N or Line	Moves to the next (older) frame. With argument, moves down *n* frames.
Meta-N	Moves to next frame and displays it like Meta-L. With argument, move down *n* frames.
Control-Meta-N	Moves to next frame even if it is "uninteresting" or still accumulating arguments. With argument, moves down *n* frames.
Control-P or Return	Moves up to previous (newer) frame. With argument, moves up *n* frames.
Meta-P	Moves to previous frame and displays it like Meta-L. With argument, moves up *n* frames.
Control-Meta-P	Moves to previous frame even if it is "uninteresting" or still accumulating arguments. With argument, moves up *n* frames.
Control-R	Returns a value or values from the current frame.
Meta-R	Reinvokes the function in the current frame (restart its execution at the beginning), optionally changing the arguments.
Control-Meta-R	Reinvokes the function in the current frame with the same arguments.
Control-S	Searches for a frame containing a specified function.
Meta-S	Reads the name of a special variable and returns that variable's value in the current frame. Instance variables of self may also be specified even if not special.
Control-Meta-S	Prints a list of special variables bound by the current frame and the values they are bound to by the frame. If the frame binds self, all the instance variables of self are listed even if they are not special.
Control-T	Throws a value to a tag.

Control-Meta-U	Moves down the stack to the previous "interesting" frame.
Control-X	Toggles the flag in the current frame that causes a trap on exit or throw through that frame.
Meta-X	Sets the flag causing a trap on exit or throw through the frame for the current frame and all the frames outside of it.
Control-Meta-X	Clears the flag causing a trap on exit or throw through the frame for the current frame and all the frames outside of it.
Control-Meta-V	Sets * to the *n*th value being returned from the current frame. This is non-nil only in a trap on exit from the frame.
Control-Meta-W	Switches to the window-oriented debugger.
Control-Z or Abort	Aborts the computation and throw back to the most recent break or debugger, to the program's command level, or to Lisp top level.
? or Help	Prints debugger command self-documentation.
Meta-<	Moves to the frame in which the error was signaled, and makes it current once again.
Meta->	Moves to the outermost (oldest) stack frame.
Control-0 through Control-Meta-9	Numeric arguments to the following command are specified by typing a decimal number with Control and/or Meta held down.
Super-A, etc.	The commands Super-A, Super-B, etc. are assigned to all the available proceed types for this error. The assignments are different each time the debugger is entered, so it prints a list of them when it starts up. Help P prints the list again.

30.7.5 Deexposed Windows and Background Processes

If the debugger is entered in a window that is not exposed, a notification is used to inform you that it has happened.

In general, a notification appears as a brief message printed inside square brackets if the selected window can print it. Otherwise, blinking text appears in the mouse documentation line telling you that a notification is waiting; to see the notification, type Terminal N or select a window that can print it. In either case, an audible beep is made.

In the case of a notification that the debugger is waiting to use a deexposed window, you can select and expose the window in which the error happened by typing Terminal 0 S. You can do this even if the notification has not been printed yet because the selected window cannot print it. If you select the waiting window, in this way or in any other way, the notification is discarded since you already know what it was intended to tell you.

If the debugger is entered in a process that has no window or other suitable stream to type out on, the window system assigns it a "background window". Since this window is initially not exposed, a notification is printed as above and you must use Terminal 0 S to see the window.

If an error happens in the scheduler stack group or the first level error handler stack group which are needed for processes to function, or in the keyboard or mouse process (both needed for the window system to function), the debugger uses the *cold load stream*, a primitive facility for terminal I/O which bypasses the window system.

If an error happens in another process but the window system is locked so that the notification mechanism cannot function, the cold load stream is used to ask what to do. You can tell the debugger to use the cold load stream immediately, to forcibly clear the window system locks and notify immediately as above, or to wait for the locks to become unlocked and then notify as above. If you tell it to wait, you can resume operation of the machine. Meanwhile, you can use the command Terminal Control-Clear-Input to forcibly unlock the locks, or Terminal Call to tell the debugger to use the cold load stream. The latter command normally enters a break-loop that uses the cold-load stream, but if there are any background errors, it offers to enter the debugger to handle them. You can also handle the errors in a Lisp listen loop of your choice by means of the function **eh** (page 727), assuming you can select a functioning Lisp listen loop.

30.7.6 Debugging after a Warm Boot

After a warm boot, the process that was running at the time of booting (or at the time the machine crashed prior to booting) may be debugged if you answer 'no' when the system asks whether to reset that process.

si:debug-warm-booted-process
Invoke the debugger, like the function **eh** (page 727), on the process that was running as of the last warm boot (assuming there was such a process).

On the CADR, the state you see in the debugger is not correct; some of the information dates from some period of time in advance of the boot or the crash.

On the Lambda, the state you see in the debugger will, in some system version, be accurate.

30.8 Tracing Function Execution

The trace facility allows the user to *trace* some functions. When a function is traced, certain special actions are taken when it is called and when it returns. The default tracing action is to print a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and value(s).

The trace facility is closely compatible with Maclisp. You invoke it through the **trace** and **untrace** special forms, whose syntax is described below. Alternatively, you can use the trace system by clicking Trace in the system menu, or by using the Meta-X Trace command in the editor. This allows you to select the trace options from a menu instead of having to remember the following syntax.

trace *Special form*

A trace form looks like:

```
        (trace spec-1 spec-2 ...)
```

Each *spec* can take any of the following forms:

a symbol
: This is a function name, with no options. The function is traced in the default way, printing a message each time it is called and each time it returns.

a list (*function-name option-1 option-2 ...*)
: *function-name* is a symbol and the *options* control how it is to be traced. The various options are listed below. Some options take arguments, which should be given immediately following the option name.

a list (:function *function-spec option-1 option-2 ...*)
: This is like the previous form except that *function-spec* need not be a symbol (see section 11.2, page 223). It exists because if *function-name* was a list in the previous form, it would instead be interpreted as the following form:

a list ((*function-1 function-2...*) *option-1 option-2 ...*)
: All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

The following trace options exist:

:break *pred*
: Causes a breakpoint to be entered after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-nil. During the breakpoint, the symbol arglist is bound to a list of the arguments of the function.

:exitbreak *pred*
: This is just like break except that the breakpoint is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the breakpoint, the symbol arglist is bound to a list of the arguments of the function, and the symbol values is bound to a list of the values that the function is returning.

:error
: Causes the error handler to be called when the function is entered. Use Resume (or Control-C) to continue execution of the function. If this option is specified, there is no printed trace output other than the error message printed by the error handler. This is semi-obsolete, as breakon is more convenient and does more exactly the right thing.

:step
: Causes the function to be single-stepped whenever it is called. See the documentation on the step facility, section 30.11, page 746.

:stepcond *pred*
: Causes the function to be single-stepped only if *pred* evaluates to non-nil.

:entrycond *pred*
: Causes trace information to be printed on function entry only if *pred* evaluates to non-nil.

:exitcond *pred*
: Causes trace information to be printed on function exit only if *pred* evaluates to non-nil.

:cond *pred*	This specifies both :exitcond and :entrycond together.
:wherein *function*	Causes the function to be traced only when called, directly or indirectly, from the specified function *function*. One can give several trace specs to **trace**, all specifying the same function but with different **wherein** options, so that the function is traced in different ways when called from different functions. This is different from **advise-within**, which only affects the function being advised when it is called directly from the other function. The **trace** :wherein option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, etc.
:argpdl *pdl*	Specifies a symbol *pdl*, whose value is initially set to nil by **trace**. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments is consed onto the *pdl* when the function is entered, and cdr'ed back off when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl may serve several functions.
:entryprint *form*	The *form* is evaluated and the value is included in the trace message for calls to the function. You can give this option multiple times, and all the *form*'s thus specified are evaluated and printed. \\ precedes the values to separate them from the arguments.
:exitprint *form*	The *form* is evaluated and the value is included in the trace message for returns from the function. You can give this option multiple times, and all the *form*'s thus specified are evaluated and printed. \\ precedes the values to separate them from the returned values.
:print *form*	The *form* is evaluated and the value is included in the trace messages for both calls to and returns from the function. Equivalent to :exitprint and :entryprint at once.
:entry *list*	This specifies a list of arbitrary forms whose values are to be printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \\ to separate it from the other information.
:exit *list*	This is similar to **entry**, but specifies expressions whose values are printed with the exit-trace. Again, the list of values printed is preceded by \\.
:arg :value :both nil	These specify which of the usual trace printouts should be enabled. If :arg is specified, then on function entry the name of the function and the values of its arguments will be printed. If :value is specified, then on function exit the returned value(s) of the function will be printed. If :both is specified, both of these will be printed. If nil is specified, neither will be printed. If none of these four options are specified the default is to :both. If any further *options* appear after one of these, they are not treated as options! Rather, they are considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function,

along with the normal trace information. The values printed will be preceded by a //, and follow any values specified by :entry or :exit.

Note that since these options "swallow" all following options, if one is given it should be the last option specified.

In the evaluation of the expression arguments to various trace options such as :cond and :break, the value of arglist is a list of the arguments given to the traced function. Thus

```
    (trace (foo :break (null (car arglist))))
```

would cause a break in foo if and only if the first argument to foo is nil. If the :break option is used, the variable arglist is valid inside the break-loop. If you setq arglist before actual function execution, the arguments seen by the function will change.

In the evaluation of the expression arguments to various trace options such as :cond and :break on exit from the traced function, the variable values is bound to a list of the resulting values of the traced function. If the :exitbreak option is used, the variables values and arglist are valid inside the break-loop. If you setq values, the values returned by the function will change.

The trace specifications may be "factored", as explained above. For example,

```
    (trace ((foo bar) :break (bad-p arglist) :value))
```

is equivalent to

```
    (trace (foo :break (bad-p arglist) :value)
           (bar :break (bad-p arglist) :value))
```

Since a list as a function name is interpreted as a list of functions, non-atomic function names (see section 11.2, page 223) are specified as follows:

```
    (trace (:function (:method flavor :message) :break t))
```

trace returns as its value a list of names of all functions it traced. If called with no arguments, as just (trace), it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, trace calls untrace before setting up the new trace.

Tracing is implemented with encapsulation (see section 11.9, page 244), so if the function is redefined (e.g. with defun or by loading it from a QFASL file) the tracing will be transferred from the old definition to the new definition.

Tracing output is printed on the stream that is the value of *trace-output*. This is synonymous with *terminal-io* unless you change it.

untrace *Special form*

Undoes the effects of trace and restores functions to their normal, untraced state. untrace accepts multiple specifications, e.g. (untrace foo quux fuphoo). Calling untrace with no arguments will untrace all functions currently being traced.

trace-compile-flag *Variable*

If the value of **trace-compile-flag** is non-nil, the functions created by **trace** are compiled, allowing you to trace special forms such as **cond** without interfering with the execution of the tracing functions. The default value of this flag is **nil**.

See also the function **compile-encapsulations**, page 302.

30.9 Breakon

The function **breakon** allows you to request that the debugger be entered whenever a certain function is called.

breakon *function-spec* &optional *condition-form*

Encapsulates the definition of *function-spec* so that a trap-on-call occurs when it is called. This enters the debugger. A trap-on-exit will occur when the stack frame is exited.

If *condition-form* is non-nil, its value should be a form to be evaluated each time *function-spec* is called. The trap occurs only if *condition-form* evaluates to non-nil. Omitting the *condition-form* is equivalent to supplying t. If **breakon** is called more than once for the same *function-spec* and different *condition-form*s, the trap occurs if any of the conditions are true.

breakon with no arguments returns a list of the functions that are broken on.

Conditional breakons are useful for causing the trap to occur only in a certain stack group. This sometimes allows debugging of functions that are being used frequently in background processes.

```
    (breakon 'foo '(eq current-stack-group ',current-stack-group))
```

If you wish to trap on calls to **foo** when called from the execution of **bar**, you can use (si:function-active-p 'bar) as the condition. If you want to trap only calls made directly from **bar**, the thing to do is

```
    (breakon '(:within bar foo))
```

rather than a conditional breakon.

To break only the *n*'th time **foo** is called, do

```
    (defvar i n)
    (breakon 'foo '(zerop (decf i)))
```

Another useful form of conditional breakon allows you to control trapping from the keyboard:

```
    (breakon 'foo '(tv:key-state :mode-lock))
```

The trap occurs only when the Mode-Lock key is down. This key is not normally used for much else. With this technique, you can successfully trap on functions used by the debugger!

unbreakon *function-spec* &optional *conditional-form*

Remove the **breakon** set on *function-spec*. If *conditional-form* is specified, remove only that condition. Breakons with other conditions are not removed.

With no arguments, **unbreakon** removes all breakons from all functions.

To cause the encapsulation which implements the breakon to be compiled, call **compile-encapsulations** or set **compile-encapsulations-flag** non-nil. See page 302. This may eliminate some of the problems that occur if you breakon a function such as **prog** that is used by the evaluator. (A conditional to trap only in one stack group will help here also.)

30.10 Advising a Function

To advise a function is to tell it to do something extra in addition to its actual definition. It is done by means of the function **advise**. The something extra is called a piece of advice, and it can be done before, after, or around the definition itself. The advice and the definition are independent, in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function to give the user information about when and how the function is called and when and with what value it returns. Advising is intended for semi-permanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing out a change to a function in a way that is easy to retract. In this case, you would call **advise** from the terminal. It can also be used for customizing a function that is part of a program written by someone else. In this case you would be likely to put a call to **advise** in one of your source files or your login init file (see page 801), rather than modifying the other person's source code.

Advising is implemented with encapsulation (see section 11.9, page 244), so if the function is redefined (e.g. with **defun** or by loading it from a QFASL file) the advice will be transferred from the old definition to the new definition.

advise *Macro*

A function is advised by the special form
`(advise` *function class name position*
form1 form2`...)`

None of this is evaluated. *function* is the function to put the advice on. It is usually a symbol, but any function spec is allowed (see section 11.2, page 223). The *forms* are the advice; they get evaluated when the function is called. *class* should be either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). The meaning of **:around** advice is explained a couple of sections below.

name is used to keep track of multiple pieces of advice on the same function. *name* is an arbitrary symbol that is remembered as the name of this particular piece of advice. If you have no name in mind, use nil; then we say the piece of advice is anonymous. A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first. Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

position says where to put this piece of advice in relation to others of the same class already present on the same function. If *position* is *nil*, the new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.

If you wish to specify the position, *position* can be the numerical index of which existing piece of advice to insert this one before. Zero means at the beginning; a very large number means at the end. Or, *position* can be the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

For example,
```
    (advise factorial :before negative-arg-check nil
      (if (minusp (first arglist))
          (ferror nil "factorial of negative argument")))
```
This modifies the (hypothetical) factorial function so that if it is called with a negative argument it signals an error instead of running forever.

advise with no arguments returns a list of advised functions.

unadvise *Macro*

(unadvise *function class position*)

removes pieces of advice. None of its arguments are evaluated. *function* and *class* have the same meaning as they do in the function advise. *position* specifies which piece of advice to remove. It can be the numeric index (zero means the first one) or it can be the name of the piece of advice.

If some of the arguments are missing or nil, all pieces of advice which match the non-nil arguments are removed. Thus, if *function* is missing or nil, all advice on all functions which match the specified *class* and *position* are removed. If *position* is missing or nil, then all advice of the specified class on the specified function is removed. If only *function* is non-nil, all advice on that function is removed.

The following are the primitive functions for adding and removing advice. Unlike the above special forms, these are functions and can be conveniently used by programs. advise and unadvise are actually macros that expand into calls to these two.

si:advise-1 *function class name position forms*
Adds advice. The arguments have the same meaning as in **advise**. Note that the *forms* argument is *not* a **&rest** argument.

si:unadvise-1 &optional *function class position*
Removes advice. If *function* or *class* or *position* is **nil** or unspecified, advice is removed from all functions or all classes of advice or advice at all positions are removed.

You can find out manually what advice a function has with **grindef**, which grinds the advice on the function as forms that are calls to **advise**. These are in addition to the definition of the function.

To cause the advice to be compiled, call **compile-encapsulations** or set **compile-encapsulations-flag** non-nil. See page 302.

30.10.1 Designing the Advice

For advice to interact usefully with the definition and intended purpose of the function, it must be able to interface to the data flow and control flow through the function. We provide conventions for doing this.

The list of the arguments to the function can be found in the variable **arglist**. **:before** advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with
```
    (setq arglist (copylist arglist))
```
After the function's definition has been executed, the list of the values it returned can be found in the variable **values**. **:after** advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a **block nil** so any piece of advice can exit the entire function with **return**. The arguments of the **return** are returned as the values of the function and no further advice is executed. If a piece of **:before** advice does this then the function's definition is not even called.

30.10.2 :around Advice

A piece of **:before** or **:after** advice is executed entirely before or entirely after the definition of the function. **:around** advice is wrapped around the definition; that is, the call to the original definition of the function is done at a specified place inside the piece of **:around** advice. You specify where by putting the symbol **:do-it** in that place.

For example, **(+ 5 :do-it)** as a piece of **:around** advice would add 5 to the value returned by the function. This could also be done by **(setq values (list (+ 5 (car values))))** as **:after** advice.

When there is more than one piece of **:around** advice, the pieces are stored in a sequence just like **:before** and **:after** advice. Then, the first piece of advice in the sequence is the one started first. The second piece is substituted for **:do-it** in the first one. The third one is substituted for :do-it in the second one.* The original definition is substituted for **:do-it** in the last piece of advice.

:around advice can access **arglist**, but **values** is not set up until the outermost **:around** advice returns. At that time, it is set to the value returned by the **:around** advice. It is reasonable for the advice to receive the values of the **:do-it** (e.g. with **multiple-value-list**) and fool with them before returning them (e.g. with **values-list**).

:around advice can return from the **block** at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain **:do-it**. Containing two instances of :do-it may be useful under peculiar circumstances. If you are careless, the original definition may be called twice, but something like

```
(if (foo) (+ 5 :do-it) (* 2 :do-it))
```

will work reasonably.

30.10.3 Advising One Function Within Another

It is possible to advise the function **foo** only for when it is called directly from a specific other function **bar**. You do this by advising the function specifier (**:within bar foo**). That works by finding all occurrences of **foo** in the definition of **bar** and replacing them with **#:altered-foo-within-bar**. (Note that this is an uninterned symbol.) This can be done even if **bar's** definition is compiled code. The symbol **#:altered-foo-within-bar** starts off with the symbol **foo** as its definition; then the symbol **#:altered-foo-within-bar**, rather than **foo** itself, is advised. The system remembers that **foo** has been replaced inside **bar**, so that if you change the definition of **bar**, or advise it, then the replacement is propagated to the new definition or to the advice. If you remove all the advice on (**:within bar foo**), so that its definition becomes the symbol **foo** again, then the replacement is unmade and everything returns to its original state.

(**grindef bar**) prints **foo** where it originally appeared, rather than **#:altered-foo-within-bar**, so the replacement is not seen. Instead, **grindef** prints calls to **advise** to describe all the advice that has been put on **foo** or anything else within **bar**.

An alternate way of putting on this sort of advice is to use **advise-within**.

advise-within *Macro*

(advise-within *within-function function-to-advise*
class name position
forms...)

advises *function-to-advise* only when called directly from the function *within-function*. The other arguments mean the same thing as with **advise**. None of them are evaluated.

To remove advice from (**:within bar foo**), you can use **unadvise** on that function specifier. Alternatively, you can use **unadvise-within**.

unadvise-within *Macro*

(unadvise-within *within-function function-to-advise class position*)

removes advice that has been placed on (:within *within-function function-to-advise*). Any of the four arguments may be missing or nil; then that argument is unconstrained. All advice matching whichever arguments are non-nil is removed. For example, (unadvise-within foo nil :before) removes all :before-advice from anything within foo. (unadvise-within) removes all advice placed on anything within anything. By contrast, (unadvise) removes all advice, including advice placed on a function for all callers. Advice placed on a function not within another specific function is never removed by unadvise-within.

The function versions of advise-within and unadvise-within are called si:advise-within-1 and si:unadvise-within-1. advise-within and unadvise-within are macros that expand into calls to the other two.

30.11 Stepping Through an Evaluation

The Step facility gives you the ability to follow every step of the evaluation of a form, and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. If your program is doing something strange, and it isn't obvious how it's getting into its strange state, then the stepper is for you.

There are two ways to enter the stepper. One is by use of the step function.

step *form*

This evaluates *form* with single stepping. It returns the value of *form*.

For example, if you have a function named foo, and typical arguments to it might be t and 3, you could say

```
    (step '(foo t 3))
```

to evaluate the form (foo t 3) with single stepping.

The other way to get into the stepper is to use the :step option of trace (see page 738). If a function is traced with the :step option, then whenever that function is called it will be single stepped.

Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be stepped by the stepper.

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a forward arrow (→) character When a macro is expanded, the expansion is printed out preceded by a double arrow (⇔) character. When a form returns a value, the form and the values are printed out preceded by a backwards arrow (←) character; if there is more than one value being returned, an and-sign (∧) character is printed between the values. When the stepper has evaluated the args to a form and is about to apply the function, it prints a lambda (λ) because entering the lambda is the next thing to be done.

Since the forms may be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from the user. There are several commands to tell the stepper how to proceed, or to look at what is happening. The commands are:

Control-N (Next)
: Steps to the Next event, then asks for another command. Events include beginning to evaluate a form at any level or finishing the evaluation of a form at any level.

Space
: Steps to the next event at this level. In other words, continue to evaluate at this level, but don't step anything at lower levels. This is a good way to skip over parts of the evaluation that don't interest you.

Control-A (Args)
: Skips over the evaluation of the arguments of this form, but pauses in the stepper before calling the function that is the car of the form.

Control-U (Up)
: Continues evaluating until we go up one level. This is like the space command, only more so; it skips over anything on the current level as well as lower levels.

Control-X (eXit)
: Exits; finishes evaluation without any more stepping.

Control-T (Type)
: Retypes the current form in full (without truncation).

Control-G (Grind)
: Grinds (i.e. prettyprints) the current form.

Control-E (Editor)
: Switches windows, to the editor.

Control-B (Breakpoint)
: Enters a break loop from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:

step-form	the current form.
step-values	the list of returned values.
step-value	the first returned value.

If you change the values of these variables, you will affect execution.

Control-L
: Clears the screen and redisplays the last 10. pending forms (forms that are being evaluated).

Meta-L
: Like Control-L, but doesn't clear the screen.

Control-Meta-L
: Like Control-L, but redisplays all pending forms.

? or Help
: Prints documentation on these commands.

It is strongly suggested that you write some little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

30.12 Evalhook

The **evalhook** facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses **evalhook**, and usually it is the only thing that ever needs to. However, if you want to write your own stepper or something similar, this is the primitive facility that you can use to do so. The way this works is a bit hairy, but unless you need to write your own stepper you don't have to worry about it.

evalhook *Variable*
evalhook *Variable*

If the value of **evalhook** is non-**nil**, then special things happen in the evaluator. Its value is called the *hook function*. When a form (any form, even a number or a symbol) is to be evaluated, the hook function is called instead. Whatever values the hook function returns are taken to be the results of the evaluation. Both **evalhook** and **applyhook** are bound to **nil** before the hook function is actually called.

The hook function receives two arguments: the form that was to be evaluated, and the lexical environment of evaluation. These two arguments allow the hook function to perform later, if it wishes, the very same evaluation that the hook was called instead of.

applyhook *Variable*
applyhook *Variable*

If the value of **applyhook** is non-**nil**, it is called the next time the interpreter is about to apply a function to its evaluated arguments. Whatever values the apply hook function returns are taken to be the results of calling the other function. Both **evalhook** and **applyhook** are bound to **nil** before the hook function is actually called.

The hook function receives three arguments: the function that was going to be called, the list of arguments it was going to receive, and the lexical environment of evaluation. These arguments allow the hook function to perform later, if it wishes, the very same evaluation that the hook was called instead of.

When either the evalhook or the applyhook is called, both variables are bound to **nil**. They are also rebound to **nil** by **break** and by the debugger, and **setq**'ed to **nil** when errors are dismissed by throwing to the Lisp top level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on the evalhook and applyhook. They apply only to evaluation—whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function **eval**. They *do not* have any effect on compiled function references, on use of the function **apply**, or on the mapping functions.

evalhook *form evalhook applyhook* &optional *environment*

Evaluates *form* in the specified *environment*, with *evalhook* and *applyhook* in effect for all recursive evaluations of subforms of *form*. However, the *evalhook* is not called for the evaluation of *form* itself.

environment is a list which represents the lexical environment to be in effect for the evaluation of *form*. nil means an empty lexical environment, in which no lexical bindings exist. This is the environment used when eval itself is called. Aside from nil, the only reasonable way to get a value to pass for *environment* is to use the last argument passed to a hook function. You must take care not to use it after the context in which it was made is exited, because environments normally contain stack lists which become garbage after their stack frames are popped.

environment has no effect on the evaluation of a variable which is regarded as special. This is always done by examining the value cell. However, environment contains the record of the local special declarations currently in effect, so it does enter in the decision of whether a variable is special.

Here is an example of the use of **evalhook**:

```
;; This function evaluates a form while printing debugging information.
(defun hook (x)
   (terpri)
   (evalhook x 'hook-function nil))

;; Notice how this function calls evalhook to evaluate the form f,
;; so as to hook the sub-forms.
(defun hook-function (f env)
   (let ((v (multiple-value-list
               (evalhook f 'hook-function nil env))))
     (format t "form: ~S~%values: ~S~%" f v)
     (values-list v)))
```

The following output might be seen from **(hook '(cons (car '(a . b)) 'c))**:

```
form: (quote (a . b))
values: ((a . b))
form: (car (quote (a . b)))
values: (a)
form: (quote c)
values: (c)
(a . c)
```

applyhook *function list-of-args evalhook applyhook* &optional *environment*

Applies *function* to *list-of-args* in the specified *environment*, with *evalhook* and *applyhook* in effect for all recursive evaluations of subforms of *function*'s body. However, *applyhook* is not called for this application of function itself. For more information, refer to the definition of **evalhook**, immediately above.

30.13 The MAR

The MAR facility allows any word or contiguous set of words to be monitored constantly, and can cause an error if the words are referenced in a specified manner. The name MAR is from the similar device on the ITS PDP-10's; it is an acronym for 'Memory Address Register'. The MAR checking is done by the Lisp Machine's memory management hardware, so the speed of general execution is not significantly slowed down when the MAR is enabled. However, the speed of accessing pages of memory containing the locations being checked is slowed down somewhat, since every reference involves a microcode trap.

These are the functions that control the MAR:

set-mar *location cycle-type* &optional *n-words*

Sets the MAR on *n-words* words, starting at *location*. *location* may be any object. Often it will be a locative pointer to a cell, probably created with the locf special form. *n-words* currently defaults to 1, but eventually it may default to the size of the object. *cycle-type* says under what conditions to trap. :read means that only reading the location should cause an error, :write means that only writing the location should, t means that both should. To set the MAR to detect setq (and binding) of the variable foo, use

```
        (set-mar (variable-location foo) :write)
```

clear-mar

Turns off the MAR. Warm-booting the machine disables the MAR but does not turn it off, i.e. references to the MARed pages are still slowed down. clear-mar does not currently speed things back up until the next time the pages are swapped out; this may be fixed some day.

mar-mode

(mar-mode) returns a symbol indicating the current state of the MAR. It returns one of:

nil	The MAR is not set.
:read	The MAR will cause an error if there is a read.
:write	The MAR will cause an error if there is a write.
t	The MAR will cause an error if there is any reference.

Note that using the MAR makes the pages on which it is set somewhat slower to access, until the next time they are swapped out and back in again after the MAR is shut off. Also, use of the MAR currently breaks the read-only feature if those pages were read-only.

Proceeding from a MAR break allows the memory reference that got an error to take place, and continues the program with the MAR still effective. When proceeding from a write, you have the choice of whether to allow the write to take place or to inhibit it, leaving the location with its old contents.

sys:mar-break (condition) *Condition*

This is the condition, not an error, signaled by a MAR break.

The condition instance supports these operations:

:object	The object one of whose words was being referenced.
:offset	The offset within the object of the word being referenced.
:value	The value read, or to be written.
:direction	Either :read or :write.

The proceed type :no-action simply proceeds, continuing with the interrupted program as if the MAR had not been set. If the trap was due to writing, the proceed type :proceed-no-write is also provided, and causes the program to proceed but does not store the value in the memory location.

Most—but not all—write operations first do a read. **setq** and **rplaca** both do. This means that if the MAR is in :read mode it catches writes as well as reads; however, they trap during the reading phase, and consequently the data to be written are not yet known. This also means that setting the MAR to **t** mode causes most writes to trap twice, first for a read and then again for a write. So when the MAR says that it trapped because of a read, this means a read at the hardware level, which may not look like a read in your program.

31. How to Read Assembly Language

Sometimes it is useful to study the machine language code produced by the Lisp Machine's compiler, usually in order to analyze an error, or sometimes to check for a suspected compiler problem. This chapter explains how the Lisp Machine's instruction set works and how to understand what code written in that instruction set is doing. Fortunately, the translation between Lisp and this instruction set is very simple; after you get the hang of it, you can move back and forth between the two representations without much trouble. The following text does not assume any special knowledge about the Lisp Machine, although it sometimes assumes some general computer science background knowledge.

31.1 Introduction

Nobody looks at machine language code by trying to interpret octal numbers by hand. Instead, there is a program called the Disassembler which converts the numeric representation of the instruction set into a more readable textual representation. It is called the Disassembler because it does the opposite of what an Assembler would do; however, there isn't actually any assembler that accepts this input format, since there is never any need to manually write assembly language for the Lisp Machine.

The simplest way to invoke the Disassembler is with the Lisp function **disassemble**. Here is a simple example. Suppose we type:

```
(defun foo (x)
  (assq 'key (get x 'propname)))

(compile 'foo)

(disassemble 'foo)
```

This defines the function **foo**, compiles it, and invokes the Disassembler to print out the textual representation of the result of the compilation. Here is what it looks like:

```
22 MOVE D-PDL FEF|6             ;'KEY
23 MOVE D-PDL ARG|0             ;X
24 MOVE D-PDL FEF|7             ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

The Disassembler is also used by the Error Handler and the Inspector. When you see stuff like the above while using one of these programs, it is disassembled code, in the same format as the **disassemble** function uses. Inspecting a compiled code object shows the disassembled code.

Now, what does this mean? Before we get started, there is just a little bit of jargon to learn.

The acronym PDL stands for Push Down List, and means the same thing as Stack: a last-in first-out memory. The terms PDL and stack will be used interchangeably. The Lisp Machine's architecture is rather typical of "stack machines"; there is a stack that most instructions deal with, and it is used to hold values being computed, arguments, and local variables, as well as flow-of-control information. An important use of the stack is to pass arguments to instructions, though not all instructions take their arguments from the stack.

The acronym 'FEF' stands for Function Entry Frame. A FEF is a compiled code object produced by the compiler. After the **defun** form above was evaluated, the function cell of the symbol **foo** contained a lambda expression. Then we compiled the function **foo**, and the contents of the function cell were replaced by a FEF. The printed representation of the FEF for **foo** looks like this:

```
#<DTP-FEF-POINTER 11464337 FOO>
```

The FEF has three parts (this is a simplified explanation): a header with various fixed-format fields; a part holding constants and invisible pointers, and the main body, holding the machine language instructions. The first part of the FEF, the header, is not very interesting and is not documented here (you can look at it with **describe** but it won't be easy to understand). The second part of the FEF holds various constants referred to by the function; for example, our function **foo** references two constants (the symbols **key** and **propname**), and so (pointers to) those symbols are stored in the FEF. This part of the FEF also holds invisible pointers to the value cells of all symbols that the function uses as variables, and invisible pointers to the function cells of all symbols that the function calls as functions. The third part of the FEF holds the machine language code itself.

Now we can read the disassembled code. The first instruction looked like this:

```
22 MOVE D-PDL FEF|6              ;'KEY
```

This instruction has several parts. The **22** is the address of this instruction. The Disassembler prints out the address of each instruction before it prints out the instruction, so that you can interpret branching instructions when you see them (we haven't seen one of these yet, but we will later). The **MOVE** is an opcode: this is a **MOVE** instruction, which moves a datum from one place to another. The **D-PDL** is a destination specification. The **D** stands for 'Destination', and so **D-PDL** means 'Destination-PDL': the destination of the datum being moved is the PDL. This means that the result will be pushed onto the PDL, rather than just moved to the top; this instruction is pushing a datum onto the stack. The next field of the instruction is **FEF|6**. This is an *address*, and it specifies where the datum is coming from. The vertical bar serves to separate the two parts of the address. The part before the vertical bar can be thought of as a *base register*, and the part after the bar can be thought of as being an offset from that register. **FEF** as a base register means the address of the FEF that we are disassembling, and so this address means the location six words into the FEF. So what this instruction does is to take the datum located six words into the FEF, and push it onto the PDL. The instruction is followed by a comment field, which looks like **;'KEY**. This is not a comment that any person wrote; the disassembler produces these to explain what is going on. The semicolon just serves to start the comment, the way semicolons in Lisp code do. In this case, the body of the comment, **'KEY**, is telling us that the address field (**FEF|6**) is addressing a constant (that is what the single-quote in **'KEY** means), and that the printed representation of that constant is **KEY**. With the help of this comment we finally get the real story about what this instruction is doing: it is pushing (a pointer to) the symbol **key** onto the stack.

The next instruction looks like this:

```
    23 MOVE D-PDL ARG|0             ;X
```

This is a lot like the previous instruction; the only difference is that a different "base register" is being used in the address. The **ARG** base register is used for addressing your arguments: **ARG|0** means that the datum being addressed is the zeroth argument. Again, the comment field explains what that means: the value of X (which was the zeroth argument) is being pushed onto the stack.

The third instruction is just like the first and second ones; it pushes the symbol **propname** onto the stack.

The fourth instruction is something new:

```
    25 (MISC) GET D-PDL
```

The first thing we see here is **(MISC)**. This means that this is one of the so-called *miscellaneous* instructions. There are quite a few of these instructions. With some exceptions, each miscellaneous instruction corresponds to a Lisp function and has the same name as that Lisp function. If a Lisp function has a corresponding miscellaneous instruction, then that function is hand-coded in Lisp Machine microcode.

Miscellaneous instructions only have a destination field; they don't have any address field. The inputs to the instruction come from the stack: the top *n* elements on the stack are used as inputs to the instruction and popped off the stack, where *n* is the number of arguments taken by the function. The result of the function is stored wherever the destination field says. In our case, the function being executed is **get**, a Lisp function of two arguments. The top two values will be popped off the stack and used as the arguments to **get** (the value pushed first is the first argument, the value pushed second is the second argument, and so on). The result of the call to **get** will be sent to the destination **D-PDL**; that is, it will be pushed onto the stack. (In case you were wondering about how we handle optional arguments and multiple-value returns, the answer is very simple: functions that use either of those features cannot be miscellaneous instructions! If you are curious as to what functions are hand-microcoded and thus available as miscellaneous instructions, you can look at the **defmic** forms in the file **SYS: SYS; DEFMIC LISP**.)

The fifth and last instruction is similar to the fourth:

```
    26 (MISC) ASSQ D-RETURN
```

What is new here is the new value of the destination field. This one is called **D-RETURN**, and it can be used anywhere destination fields in general can be used (like in **MOVE** instructions). Sending something to "Destination-Return" means that this value should be the returned value of the function, and that we should return from this function. This is a bit unusual in instruction sets; rather than having a "return" instruction, we have a destination that, when stored into, returns from the function. What this instruction does, then, is to invoke the Lisp function **assq** on the top two elements of the stack and return the result of **assq** as the result of this function.

Now, let's look at the program as a whole and see what it did:

```
22 MOVE D-PDL FEF|6              ;'KEY
23 MOVE D-PDL ARG|0              ;X
24 MOVE D-PDL FEF|7              ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

First it pushes the symbol key. Then it pushes the value of x. Then it pushes the symbol propname. Then it invokes get, which pops the value of x and the symbol propname off the stack and uses them as arguments, thus doing the equivalent of evaluating the form (get x 'propname). The result is left on the stack; the stack now contains the result of the get on top, and the symbol key underneath that. Next, it invokes assq on these two values, thus doing the equivalent of evaluating (assq 'key (get x 'propname)). Finally, it returns the value produced by assq. Now, the original Lisp program we compiled was:

```
(defun foo (x)
  (assq 'key (get x 'propname)))
```

We can see that the code produced by the compiler is correct: it will do the same thing as the function we defined will do.

In summary, we have seen two kinds of instructions so far: the MOVE instruction, which takes a destination and an address, and two of the large set of miscellaneous instructions, which take only a destination, and implicitly get their inputs from the stack. We have seen two destinations (D-PDL and D-RETURN), and two forms of address (FEF addressing and ARG addressing).

31.2 A More Advanced Example

Here is a more complex Lisp function, demonstrating local variables, function calling, conditional branching, and some other new instructions.

```
(defun bar (y)
  (let ((z (car y)))
    (cond ((atom z)
           (setq z (cdr y))
           (foo y))
          (t
           nil))))
```

The disassembled code looks like this:

```
20 CAR D-PDL ARG|0              ;Y
21 POP LOCAL|0                  ;Z
22 BR-NOT-ATOM 27
23 CDR D-PDL ARG|0              ;Y
24 POP LOCAL|0                  ;Z
25 CALL D-RETURN FEF|6          ;#'FOO
26 MOVE D-LAST ARG|0            ;Y
27 MOVE D-RETURN 'NIL
```

The first instruction here is a CAR instruction. It has the same format as MOVE: there is a destination and an address. The CAR instruction reads the datum addressed by the address, takes the car of it, and stores the result into the destination. In our example, the first instruction addresses the zeroth argument, and so it computes (car y); then it pushes the result onto the stack.

The next instruction is something new: the POP instruction. It has an address field, but it uses it as a destination rather than as a source. The POP instruction pops the top value off the stack, and stores that value into the address specified by the address field. In our example, the value on the top of the stack is popped off and stored into address LOCAL|0. This is a new form of address; it means the zeroth local variable. The ordering of the local variables is chosen by the compiler, and so it is not fully predictable, although it tends to be by order of appearance in the code; fortunately you never have to look at these numbers, because the comment field explains what is going on. In this case, the variable being addressed is z. So this instruction pops the top value on the stack into the variable z. The first two instructions work together to take the car of y and store it into z, which is indeed the first thing the function bar ought to do. (If you have two local variables with the same name, then the comment field won't tell you which of the two you're talking about; you'll have to figure that out yourself. You can tell two local variables with the same name apart by looking at the number in the address.)

The next instruction is a familiar MOVE instruction, but it uses a new destination: D-IGNORE. This means that the datum being addressed isn't moved anywhere. If so, then why bother doing this instruction? The reason is that there is conceptually a set of *indicator* bits, as are found in most modern computers such as the 68000, the Vax, as well as in obsolete computers such as the 370. Every instruction that moves or produces a datum sets the indicator bits from that datum so that following instructions can test them. So the reason that the MOVE instruction is being done is so that someone can test the indicators set up by the value that was moved, namely the value of z.

All instructions except the branch instructions set the indicator bits from the result produced and/or stored by that instruction.

The next instruction is a conditional branch; it changes the flow of control, based on the values in the indicator bits, which in this case reflect the value popped by the POP instruction 21. The branch instruction is BR-NOT-ATOM 27, which means "Branch, if the quantity was not an atom, to location 27; otherwise proceed with execution". If z was an atom, the Lisp Machine branches to location 27, and execution proceeds there. (As you can see by skipping ahead, location 27 just contains a MOVE instruction, which will cause the function to return nil.)

If z is not an atom, the program keeps going, and the CDR instruction is next. This is just like the CAR instruction except that it takes the cdr; this instruction pushes the value of (cdr y) onto the stack. The next one pops that value off into the variable z.

There are just two more instructions left. These two instructions are our first example of how function calling is compiled. It is the only really tricky thing in the instruction set. Here is how it works in our example:

```
25 CALL D-RETURN FEF|6          ;#'FOO
26 MOVE D-LAST ARG|0            ;Y
```

The form being compiled here is (foo y). This means we are applying the function which is in the function cell of the symbol foo, and passing it one argument, the value of y. The way function calling works is in the following three steps. First of all, there is a CALL instruction that specifies the function object being applied to arguments. This creates a new stack frame on the stack, and stores the function object there. Secondly, all the arguments being passed except the last one are pushed onto the stack. Thirdly and lastly, the last argument is sent to a special destination, called D-LAST, meaning "this is the last argument". Storing to this destination is what actually calls the function, *not* the CALL instruction itself.

There are two things you might wonder about this. First of all, when the function returns, what happens to the returned value? Well, this is what we use the destination field of the CALL instruction for. The destination of the CALL is not stored into at the time the CALL instruction is executed; instead, it is saved on the stack along with the function operation (in the stack frame created by the CALL instruction). Then, when the function actually returns, its result is stored into that destination.

The other question is what happens when there isn't any last argument; that is, when there is a call with no arguments at all? This is handled by a special instruction called CALL0. The address of CALL0 addresses the function object to be called; the call takes place immediately and the result is stored into the destination specified by the destination field of the CALL0 instruction.

So, let's look at the two-instruction sequence above. The first instruction is a CALL; the function object it specifies is at FEF|6, which the comment tells us is the contents of the function cell of foo (the FEF contains an invisible pointer to that function cell). The destination field of the CALL is D-RETURN, but we aren't going to store into it yet; we will save it away in the stack frame and use it later. So the function doesn't return at this point, even though it says D-RETURN in the instruction; this is the tricky part.

Next we have to push all the arguments except the last one. Well, there's only one argument, so nothing needs to be done here. Finally, we move the last argument (that is, the only argument: the value of y) to D-LAST, using the MOVE instruction. Moving to D-LAST is what actually invokes the function, so at this point the function foo is invoked. When it returns, its result is sent to the destination stored in the stack frame: D-RETURN. Therefore, the value returned by the call to foo will be returned as the value of the function bar. Sure enough, this is what the original Lisp code says to do.

When the compiler pushes arguments to a function call, it sometimes does it by sending the values to a destination called D-NEXT (meaning the "next" argument). This is exactly the same as D-PDL when producing a compiled function. The distinction is important when the compiler output is passed to the microcompiler to generate microcode.

Here is another example to illustrate function calling. This Lisp function calls one function on the results of another function.

```
(defun a (x y)
  (b (c x y) y))
```

The disassembled code looks like this:

```
22 CALL D-RETURN FEF|6          ;#'B
23 CALL D-PDL FEF|7             ;#'C
24 MOVE D-PDL ARG|0             ;X
25 MOVE D-LAST ARG|1            ;Y
26 MOVE D-LAST ARG|1            ;Y
```

The first instruction starts off the call to the function **b**. The destination field is saved for later: when this function returns, we will return its result as **a**'s result. Next, the call to **c** is started. Its destination field, too, is saved for later; when **c** returns, its result should be pushed onto the stack, so that it will be the next argument to **b**. Next, the first and second arguments to **c** are passed; the second one is sent to D-LAST and so the function **c** is called. Its result, as we said, will be pushed onto the stack, and thus become the first argument to **b**. Finally, the second argument to **b** is passed, by storing in D-LAST; **b** gets called, and its result is sent to D-RETURN and is returned from **a**.

31.3 The Rest of the Instructions

Now that we've gotten some of the feel for what is going on, I will start enumerating the instructions in the instruction set. The instructions fall into four classes. Class I instructions have both a destination and an address. Class II instructions have an address, but no destination. Class III instructions are the branch instructions, which contain a branch address rather than a general base-and-offset address. Class IV instructions have a destination, but no address; these are the miscellaneous instructions.

We have already seen just about all the Class I instructions. There are nine of them in all: MOVE, CALL, CALL0, CAR, CDR, CAAR, CADR, CDAR, and CDDR. MOVE just moves a datum from an address to a destination; the CxR and CxxR instructions are the same but perform the function on the value before sending it to the destination; CALL starts off a call to a function with some arguments; CALL0 performs a call to a function with no arguments.

We've seen most of the possible forms of address. So far we have seen the FEF, ARG, and LOCAL base registers. There are two other kinds of addresses. One uses a "constant" base register, which addresses a set of standard constants: NIL, T, 0, 1, and 2. The disassembler doesn't even bother to print out CONSTANT|*n*, since the number *n* would not be even slightly interesting; it just prints out 'NIL or '1 or whatever. The other kind of address is a special one printed as PDL-POP, which means that to read the value at this address, an object should be popped off the top of the stack.

There are more Class II instructions. The only one we've seen so far is POP, which pops a value off the stack and stores it into the specified address. Another, called MOVEM (from the PDP-10 opcode name, meaning MOVE to Memory), stores the top element of the stack into the specified address, but doesn't pop it off the stack.

Seven Class II instructions implement heavily-used two-argument functions: +, -, *, /, LOGAND, LOGXOR, and LOGIOR. These instructions take the first argument from the top of the stack (popping it off) and their second argument from the specified address, and they push the result on the stack. Thus the stack level does not change due to these instructions. Here is a small function that shows some of these new things:

```
(defun foo (x y)
  (setq x (logxor y (- x 2))))
```

The disassembled code looks like this:

```
16 MOVE D-PDL ARG|1            ;Y
17 MOVE D-PDL ARG|0            ;X
20 - '2
21 LOGXOR PDL-POP
22 MOVEM ARG|0                 ;X
23 MOVE D-RETURN PDL-POP
```

Instructions 20 and 21 use two of the new Class II instructions: the - and LOGXOR instructions. Instructions 21 and 23 use the PDL-POP address type, and instruction 20 uses the "constant" base register to get to a fixnum 2. Finally, instruction 22 uses the MOVEM instruction; the compiler wants to use the top value of the stack to store it into the value of **x**, but it doesn't want to pop it off the stack because it has another use for it: to return it from the function.

Another four Class II instructions implement some commonly used predicates: =, >, <, and EQ. The two arguments come from the top of the stack and the specified address; the stack is popped, the predicate is applied to the two objects, and the result is left in the indicators so that a branch instruction can test it, and branch based on the result of the comparison. These instructions remove the top item on the stack and don't put anything back, unlike the previous set, which put their results back on the stack.

Next, there are four Class II instructions to read, modify, and write a quantity in ways that are common in Lisp code. These instructions are called SETE-CDR, SETE-CDDR, SETE-1+, and SETE-1-. The SETE- means to set the addressed value to the result of applying the specified one-argument function to the present value. For example, SETE-CDR means to read the value addressed, apply **cdr** to it, and store the result back in the specified address. This is used when compiling **(setq x (cdr x))**, which commonly occurs in loops; the other functions are used frequently in loops, too.

There are two instructions used to bind special variables. The first is **BIND-NIL**, which binds the cell addressed by the address field to **nil**; the second is **BIND-POP**, which binds the cell to an object popped off the stack rather than **nil**. The latter instruction pops a value off the stack; the former does not use the stack at all.

There are two instructions to store common values into addressed cells. **SET-NIL** stores **nil** into the cell specified by the address field; **SET-ZERO** stores **0**. Neither instruction uses the stack at all.

Finally, the **PUSH-E** instruction creates a locative pointer to the cell addressed by the specified address, and pushes it onto the stack. This is used in compiling **(value-cell-location 'z)** where **z** is an argument or a local variable, rather than a symbol (special variable).

Those are all of the Class II instructions. Here is a contrived example that uses some of the ones we haven't seen, just to show you what they look like:

```
(defun weird (x y)
  (cond ((= x y)
         (let ((*foo* nil) (*bar* 5))
           (declare (special *foo* *bar*))
           (setq x (cdr x)))
         nil)
        (t
         (setq x nil)
         (caar (variable-location y)))))
```

The disassembled code looks like this:

```
24 MOVE D-PDL ARG|0          ;X
25 = ARG|1                   ;Y
26 BR-NIL 35
27 BIND-NIL FEF|6            ;*FOO*
30 PUSH-NUMBER 5
31 BIND-POP FEF|7            ;*BAR*
32 SETE-CDR ARG|0            ;X
33 (MISC) UNBIND 2 bindings
34 MOVE D-RETURN 'NIL
35 SET-NIL ARG|0             ;X
36 PUSH-E ARG|1              ;Y
37 CAAR D-RETURN PDL-POP
```

Instruction 25 is an **=** instruction; it numerically compares the top of the stack, **x**, with the addressed quantity, **y**. The **x** is popped off the stack, and the indicators are set to the result of the equality test. Instruction 26 checks the indicators, branching to 35 if the result of the call to **=** was **nil**; that is, the machine will branch to 35 if the two values were not equal. Instruction 27 binds ***foo*** to **nil**; instructions 30 and 31 bind ***bar*** to **5**. Instruction 30 is a peculiar class IV instruction called **PUSH-NUMBER** which pushes a constant integer on the stack. The integer must be in the range of zero to 511 in order for **PUSH-NUMBER** to be used. Instruction 32 demonstrates the use of **SETE-CDR** to compile **(setq x (cdr x))**, and instruction 35 demonstrates the use of SET-NIL to compile (setq x nil). Instruction 36 demonstrates the use of PUSH-E to compile (variable-location y).

The Class III instructions are for branching. These have neither addresses nor destinations of the usual sort. Instead, they have branch-addresses; they say where to branch, if the branch is going to happen. There are several instructions, differing in the conditions under which they branch and whether they pop the stack. Branch-addresses are stored internally as self-relative addresses, to make Lisp Machine code relocatable, but the disassembler does the addition for you and prints out FEF-relative addresses so that you can easily see where the branch is going to.

The branch instructions we have seen so far decide whether to branch on the basis of the nil-indicator, that is, whether the last value dealt with was nil or non-nil. BR-NIL branches if it was nil, and BR-NOT-NIL branches if it was not nil. There are two more instructions that test the result of the atom predicate on the last value dealt with. BR-ATOM branches if the value was an atom (that is, if it was anything besides a cons), and BR-NOT-ATOM branches if the value was not an atom (that is, if it was a cons). The BR instruction is an unconditional branch (it always branches).

None of the above branching instructions deal with the stack. There are two more instructions called BR-NIL-POP and BR-NOT-NIL-POP, which are the same as BR-NIL and BR-NOT-NIL except that if the branch is not done, the top value on the stack is popped off the stack. These are used for compiling and and or special forms.

Finally, there are the Class IV instructions, most of which are miscellaneous hand-microcoded Lisp functions. The file SYS: SYS; DEFMIC LISP has a list of all the miscellaneous instructions. Most correspond to Lisp functions, including the subprimitives, although some of these functions are very low level internals that may not be documented anywhere (don't be disappointed if you don't understand all of them). Please do not look at this file in hopes of finding obscure functions that you think you can use to speed up your programs; in fact, the compiler automatically uses these things when it can, and directly calling weird internal functions will only serve to make your code hard to read, without making it any faster. In fact, we don't guarantee that calling undocumented functions will continue to work in the future.

The DEFMIC file can be useful for determining if a given function is in microcode, although the only definitive way to tell is to compile some code that uses it and look at the results, since sometimes the compiler converts a documented function with one name into an undocumented one with another name.

31.4 Function Entry

When a function is first entered in the Lisp Machine, interesting things can happen because of the features that are invoked by use of the various lambda-list keywords. The microcode performs various services when a function is entered, even before the first instruction of the function is executed. These services are called for by various fields of the header portion of the FEF, including a list called the *Argument Descriptor List*, or *ADL*. We won't go into the detailed format of any of this, as it is complex and the details are not too interesting. Disassembling a function that makes use of the ADL prints a summary of what the ADL says to do, before the beginning of the code.

The function-entry services include the initialization of unsupplied optional arguments and of &AUX variables. The ADL has a little instruction set of its own, and if the form that computes the initial value is something simple, such as a constant or a variable, then the ADL can handle things itself. However, if things get too complicated, instructions are needed, and the compiler generates some instructions at the front of the function to initialize the unsupplied variables. In this case, the ADL specifies several different starting addresses for the function, depending on which optional arguments have been supplied and which have been omitted. If all the optional arguments are supplied, then the ADL starts the function off after all the instructions that would have initialized the optional arguments; since the arguments were supplied, their values should not be set, and so all these instructions are skipped over. Here's an example:

```
(defvar *y*)

(defun foo (&optional (x (car *y*)) (z (* x 3)))
  (cons x z))
```

The disassembled code looks like this:

```
Arg 0 (X) is optional, local,
  initialized by the code up to pc 32.
Arg 1 (Z) is optional, local,
  initialized by the code up to pc 35.

30 CAR D-PDL FEF|6              ;*Y*
31 POP ARG|0                    ;X
32 MOVE D-PDL ARG|0             ;X
33 * '3
34 POP ARG|1                    ;Z
35 MOVE D-PDL ARG|0             ;X
36 MOVE D-PDL ARG|1             ;Z
37 (MISC) CONS D-RETURN
```

If no arguments are supplied, the function will be started at instruction 30; if only one argument is supplied, it will be started at instruction 32; if both arguments are supplied, it will be started at instruction 35.

The thing to keep in mind here is that when there is initialization of variables, you may see it as code at the beginning of the function, or you may not, depending upon whether it is too complex for the ADL to handle. This is true of **&aux** variables as well as unsupplied **&optional** arguments.

When there is a **&rest** argument, it is passed to the function as the zeroth local variable, rather than as any of the arguments. This is not really so confusing as it might seem, since a **&rest** argument is not an argument passed by the caller; rather it is a list of some of the arguments, created by the function-entry microcode services. In any case the comment tells you what is going on. In fact, one hardly ever looks much at the address fields in disassembled code, since the comment tells you the right thing anyway. Here is a silly example of the use of a **&rest** argument:

```
(defun prod (&rest values)
  (apply #'* values))
```

The disassembled code looks like this:

```
20 MOVE D-PDL FEF|6              ;#'*
21 MOVE D-PDL LOCAL|0            ;VALUES
22 (MISC) APPLY D-RETURN
```

As can be seen, values is referred to as LOCAL|0.

Another thing the microcode does at function entry is to bind the values of any arguments or &aux variables that are special. Thus, you won't see any BIND instructions for binding them.

31.5 Special Class IV Instructions

We said earlier that most of the Class IV instructions are miscellaneous hand-microcoded Lisp functions. However, a few of them are not Lisp functions at all. There are two instructions that are printed as UNBIND 3 bindings or POP 7 values; the number can be anything up to 16 (these numbers are printed in decimal). These instructions just do what they say, unbinding the last *n* values that were bound or popping the top *n* values off the stack.

Another Class IV instruction is PUSH-NUMBER. It pushes a constant integer, in the range zero to 511. An example of it appeared on page 760.

The array referencing functions—aref, aset, and aloc—take a variable number of arguments, but they are handled differently depending on how many there are. For one-, two-, and three-dimensional arrays, these functions are turned into internal functions with names ar-1, as-1, and ap-1 (with the number of dimensions substituted for 1). Again, there is no point in using these functions yourself; it would only make your code harder to understand but not any faster at all. When there are more than three dimensions, the functions aref, aset and aloc are called in the ordinary manner.

```
(defun foo (y x i j &aux v)
  (setq v (aref x i j))
  (setf (aref y i) v))

16 MOVE D-PDL ARG|1              ;X
17 MOVE D-PDL ARG|2              ;I
20 MOVE D-PDL ARG|3              ;J
21 (MISC) AR-2 D-PDL
22 POP LOCAL|0                   ;V
23 MOVE D-PDL ARG|0              ;Y
24 MOVE D-PDL ARG|2              ;I
25 MOVE D-PDL LOCAL|0            ;V
26 (MISC) SET-AR-1 D-RETURN
```

Reference to one-dimensional arrays with constant subscripts use special instructions which have the array index encoded instead of an address.

```
(defun foo (x)
  (+ (aref x 3) (array-leader x 2))
  (setf (aref x 5) t))

FOO:
16 MOVE D-PDL ARG|0            ;X
17 AR-1 (3) D-IGNORE
20 MOVE D-PDL ARG|0            ;X
21 ARRAY-LEADER (2) D-IGNORE
22 MOVE D-PDL ARG|0            ;X
23 MOVE D-PDL 'T
24 SET-AR-1 (5) D-RETURN
```

The AR-1 instruction is to be distinguished from the MISC AR-1 instruction. AR-1 pops an array off the stack and encodes the subscript itself. The 3 in (3) is the subscript. ARRAY-LEADER is similar but refers to an array leader slot. SET-AR-1 pops an array and then pops a value to store into it at the specified slot. SET-AR-1 is analogous. There also exist %INSTANCE-REF and SET-%INSTANCE-REF instructions.

When you call a function and expect to get more than one value back, a slightly different kind of function calling is used. Here is an example that uses multiple-value to get two values back from a function call:

```
(defun foo (x)
  (let (y z)
    (multiple-value (y z)
      (bar 3))
    (+ x y z)))
```

The disassembled code looks like this:

```
20 MOVE D-PDL FEF|6            ;#'BAR
21 MOVE D-PDL '2
22 (MISC) %CALL-MULT-VALUE D-IGNORE
23 MOVE D-LAST '3
24 POP LOCAL|1                 ;Z
25 POP LOCAL|0                 ;Y
26 MOVE D-PDL ARG|0            ;X
27 + LOCAL|0                   ;Y
30 + LOCAL|1                   ;Z
31 MOVE D-RETURN PDL-POP
```

A %CALL-MULT-VALUE instruction is used instead of a CALL instruction. The destination field of %CALL-MULT-VALUE is unused and will always be D-IGNORE. %CALL-MULT-VALUE takes two "arguments", which it finds on the stack; it pops both of them. The first one is the function object to be applied; the second is the number of return values that are expected.

The rest of the call proceeds as usual, but when the call returns, the returned values are left on the stack. The number of objects left on the stack is always the same as the second "argument" to %CALL-MULT-VALUE. In our example, the two values returned are left on the stack, and they are immediately popped off into z and y. There is also a %CALLO-MULT-VALUE instruction, for the same reason CALLO exists.

The multiple-value-bind form works similarly; here is an example:

```
(defun foo (x)
  (multiple-value-bind (y *foo* z)
      (bar 3)
    (declare (special *foo*))
    (+ x y z)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|7             ;#'BAR
23 MOVE D-PDL '3
24 (MISC) %CALL-MULT-VALUE D-IGNORE
25 MOVE D-LAST '3
26 POP LOCAL|1                  ;Z
27 BIND-POP FEF|6               ;*FOO*
30 POP LOCAL|0                  ;Y
31 MOVE D-PDL ARG|0             ;X
32 + LOCAL|0                    ;Y
33 + LOCAL|1                    ;Z
34 MOVE D-RETURN PDL-POP
```

The %CALL-MULT-VALUE instruction is still used, leaving the results on the stack; these results are used to bind the variables.

Calls done with multiple-value-list work with the %CALL-MULT-VALUE-LIST instruction. It takes one "argument" on the stack: the function object to apply. When the function returns, the list of values is left on the top of the stack. Here is an example:

```
(defun foo (x y)
  (multiple-value-list (bar -7 y x)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6             ;#'BAR
23 (MISC) %CALL-MULT-VALUE-LIST D-IGNORE
24 MOVE D-PDL FEF|7             ;'-7
25 MOVE D-PDL ARG|1             ;Y
26 MOVE D-LAST ARG|0            ;X
27 MOVE D-RETURN PDL-POP
```

Returning of more than one value from a function is handled by special miscellaneous instructions. %RETURN-2 and %RETURN-3 are used to return two or three values; these instructions take two and three arguments, respectively, on the stack and return from the current function just as storing to D-RETURN would. If there are more than three return values, they are all pushed, then the number that there were is pushed, and then the %RETURN-N instruction is executed. None of these instructions use their destination field. Note: the **return-list** function is just an ordinary miscellaneous instruction; it takes the list of values to return as an argument on the stack and returns those values from the current function.

The function **apply** is compiled using a special instruction called %SPREAD to iterate over the elements of its last argument, which should be a list. %SPREAD takes one argument (on the stack), which is a list of values to be passed as arguments (pushed on the stack). If the destination of %SPREAD is D-PDL (or D-NEXT), then the values are just pushed; if it is D-LAST, then after the values are pushed, the function is invoked. **apply** with more than two arguments will always compile using a %SPREAD whose destination is D-LAST. Here is an example:

```
(defun foo (a b &rest c)
  (apply #'format t a c)
  b)
```

The disassembled code looks like this:

```
FOO:
20 CALL D-IGNORE FEF|6          ;#'FORMAT
21 MOVE D-PDL 'T
22 MOVE D-PDL ARG|0             ;A
23 MOVE D-PDL LOCAL|0           ;C
24 (MISC) %SPREAD D-LAST
25 MOVE D-RETURN ARG|1          ;B
```

Note that in instruction 23, the address LOCAL|0 is used to access the **&rest** argument.

The **catch** special form is also handled specially by the compiler. Here is a simple example of **catch**:

```
(defun a ()
  (catch 'foo (bar)))
```

The disassembled code looks like this:

```
24 MOVE D-PDL FEF|6             ;'30
25 (MISC) %CATCH-OPEN D-RETURN
26 MOVE D-PDL FEF|7             ;'FOO
27 CALL0 D-RETURN FEF|8         ;#'BAR
```

The %CATCH-OPEN instruction is like the CALL instruction; it starts a call to the **catch** function. It takes one "argument" on the stack, which is the location in the program that should be branched to if this **catch** is **thrown** to. In addition to saving that program location, the instruction saves the state of the stack and of special-variable binding so that they can be restored in the event of a **throw**. So instructions 24 and 25 start a **catch** block, and the rest of the function computes the two arguments of the **catch**. Note, however, that **catch** is not actually called. The last form inside the **catch**, in this case **(bar)**, is compiled so as to return its values directly out of the function **a**. The only way that the inactive stack frame for **catch** matters is if a **throw** is done during the execution of **bar**. This searches for a pending call to **catch** and returns from that frame. In this case, since the **%CATCH-OPEN** instruction specifies **D-RETURN**, the values thrown are returned from **a**.

You may have wondered why instruction 24 is there at all. If the destination of a **catch** is not **D-RETURN**, it is necessary for **throw** to resume execution of the function containing the **catch**. Then it is necessary to specify what instruction to resume at. For example:

```
(defun a ()
  (catch 'foo (bar))
  (print t))
```

The disassembled code looks like this:

```
26 MOVE D-PDL FEF|6            ;'32
27 (MISC) %CATCH-OPEN D-IGNORE
30 MOVE D-PDL FEF|7            ;'(BAR)
31 MOVE D-LAST FEF|8           ;'FOO
32 CALL D-RETURN FEF|9         ;#'PRINT
33 MOVE D-LAST 'T
```

The instruction 26 pushes 32, which is the number of instruction at which execution should resume if there is a **throw**.

To allow compilation of **(multiple-value (...) (catch ...))**, there is a special instruction called **%CATCH-OPEN-MULT-VALUE**, which is a cross between **%CATCH-OPEN** and **%CALL-MULT-VALUE**.

32. Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read on the stream ***query-io***, which normally is synonymous with ***terminal-io*** but can be rebound to another stream for special applications.

The macro **with-timeout** (see page 686) can be used with the functions in this chapter to assume an answer if the user does not respond in a fixed period of time.

We first describe two simple functions for yes-or-no questions, then the more general function on which all querying is built.

y-or-n-p &optional *format-string* &rest *format-args*

This is used for asking the user a question whose answer is either 'y' for 'yes' or 'n' for 'no'. It prints a message by passing *format-string* and *format-args* to **format**, reads a one-character answer, echoes it as 'Yes' or 'No', and returns **t** if the answer is 'yes' or **nil** if the answer is 'no'. The characters which mean 'yes' are 'Y', 'T', **Space**, and **Hand-up**. The characters which mean "no" are 'N', **Rubout**, and **Hand-down**. If any other character is typed, the function beeps and demands a 'Y or N' answer.

You should include a question mark and a space at the end of the message. **y-or-n-p** does type '(Y or N)' for you.

query-io is used for all input and output.

y-or-n-p should be used only for questions that the user knows are coming. If the user is not going to be anticipating the question (e.g. if the question is "Do you really want to delete all of your files?" out of the blue) then **y-or-n-p** should not be used, because the user might type ahead a 'T', 'Y', 'N', **Space**, or **Rubout**, and therefore accidentally answer the question. In such cases, use **yes-or-no-p**.

yes-or-no-p &optional *format-string* &rest *format-args*

This is used for asking the user a question whose answer is either 'yes' or 'no'. It prints a message by passing *format-string* and *format-args* to **format**, beeps, and reads in a line from ***query-io***. If the line is 'yes', it returns **t**. If the line is 'no', it returns **nil**. (Case is ignored, as are leading and trailing spaces and tabs.) If the input line is anything else, **yes-or-no-p** beeps and demands a 'yes or no' answer.

You should include a question mark and a space at the end of the message. **yes-or-no-p** does type '(Yes or No)' for you.

query-io is used for all input and output.

To allow the user to answer a yes-or-no question with a single character, use **y-or-n-p**. **yes-or-no-p** should be used for unanticipated or momentous questions; this is why it beeps and why it requires several keystrokes to answer it.

fquery *options format-string* &rest *format-args*

Asks a question, printed by (format *query-io* *format-string format-args*...), and returns the answer. fquery takes care of checking for valid answers, reprinting the question when the user clears the screen, giving help, and so forth.

options is a list of alternating keywords and values, used to select among a variety of features. Most callers pass a constant list as the *options* (rather than consing up a list whose contents varies). The keywords allowed are:

:type	What type of answer is expected. The currently-defined types are :tyi (a single character), :readline or :mini-buffer-or-readline (a line terminated by a carriage return). :tyi is the default. :mini-buffer-or-readline is nearly the same as :readline, the only difference being that the former uses a minibuffer if used inside the editor.
:choices	Defines the allowed answers. The allowed forms of choices are complicated and explained below. The default is the same set of choices as the y-or-n-p function (see above). Note that the :type and :choices options should be consistent with each other.
:list-choices	If t, the allowed choices are listed (in parentheses) after the question. The default is t; supplying nil causes the choices not to be listed unless the user tries to give an answer which is not one of the allowed choices.
:help-function	Specifies a function to be called if the user hits the Help key. The default help-function simply lists the available choices. Specifying nil disables special treatment of Help. Specifying a function of three arguments—the stream, the list of choices, and the type-function—allows smarter help processing. The type-function is the internal form of the :type option and can usually be ignored.
:condition	If non-nil, a signal name (see page 713) to be signaled before asking the question. A condition handler may handle the condition, specifying an answer for fquery to return, in which case the user is not asked. The details are given below. The default signal name is :fquery, which signals condition name :fquery.
:fresh-line	If t, *query-io* is advanced to a fresh line before asking the question. If nil, the question is printed wherever the cursor was left by previous typeout. The default is t.
:beep	If t, fquery beeps to attract the user's attention to the question. The default is nil, which means not to beep unless the user tries to give an answer which is not one of the allowed choices.
:stream	The value should be either an I/O stream or a symbol or expression that will evaluate to one. fquery uses the specified stream instead of *query-io* for all its input and output.
:clear-input	If t, fquery throws away type-ahead before reading the user's response to the question. Use this for unexpected questions. The default is nil, which means not to throw away type-ahead unless the user tries to give an answer which is not one of the allowed choices. In that case, type-ahead

is discarded since the user probably wasn't expecting the question.

:make-complete
: If t and *query-io* is a typeout-window, the window is "made complete" after the question has been answered. This tells the system that the contents of the window are no longer useful. Refer to the window system documentation for further explanation. The default is t.

The argument to the :choices option is a list each of whose elements is a *choice*. The cdr of a choice is a list of the user inputs which correspond to that choice. These should be characters for :type :tyi or strings for :type :readline. The car of a choice is either a symbol which fquery should return if the user answers with that choice, or a list whose first element is such a symbol and whose second element is the string to be echoed when the user selects the choice. In the former case nothing is echoed. In most cases :type :readline would use the first format, since the user's input has already been echoed, and :type :tyi would use the second format, since the input has not been echoed and furthermore is a single character, which would not be mnemonic to see on the display.

A choice can also be the symbol :any. If used, it must be the last choice. It means that any input is allowed, and should simply be returned as a string or character if it does not match any of the other choices.

Perhaps this can be clarified by example. The yes-or-no-p function uses this list of choices:

```
((t "Yes") (nil "No"))
```

and the y-or-n-p function uses this list:

```
(((t "Yes.") #\y #\t #\space #\hand-up)
 ((nil "No.") #\n #\rubout #\hand-down))
```

If a signal name is specified (or allowed to default to :fquery), before asking the question fquery will signal it. (See section 30.1, page 698 for information about conditions.) make-condition will receive, in addition to the signal name, all the arguments given to fquery, including the list of options, the format string, and all the format arguments. fquery provides one proceed type, :new-value, and if a condition handler proceeds, the argument it proceeds with is returned by **fquery**.

If you want to use the formatted output functions instead of format to produce the promting message, write

(fquery *options* (format:outfmt *exp-or-string exp-or-string* ...))

format:outfmt puts the output into a list of a string, which makes format print it exactly as is. There is no need to supply additional arguments to the fquery unless it signals a condition. In that case the arguments might be passed so that the condition handler can see them. The condition handler will receive a list containing one string, the message, as its third argument instead of just a string. If this argument is passed along to **format**, all the right things happen.

fquery (condition) *Condition*

This condition is signaled, by default, by fquery. The condition instance supports these operations:

:options	Returns the list of options given to fquery.
:format-string	Returns the format string given to fquery.
:format-args	Returns the list of additional args for format, given to fquery.

One proceed type is provided, :new-value. It should be used with a single argument, which will be returned by fquery in lieue of asking the user.

format:y-or-n-p-options *Constant*

A suitable list to pass as the first argument to fquery to make it behave like y-or-n-p.

format:yes-or-no-p-options *Constant*

A suitable list to pass as the first argument to fquery to make it behave like yes-or-no-p.

format:y-or-n-p-choices *Constant*

A list which y-or-n-p uses as the value of the :choices option.

33. Initializations

There are a number of programs and facilities in the Lisp Machine that require that "initialization routines" be run either when the facility is first loaded, or when the system is booted, or both. These initialization routines may set up data structures, start processes running, open network connections, and so on.

It is easy to perform an action when a file is loaded: simply place an expression to perform the action in the file. But this causes the action to be repeated if the file is loaded a second time, and often that should not be done. Also, this does not provide a way to cause actions to be taken at other times, such as when the system is booted or when a garbage collection is started.

The *initialization list* facility serves these needs. An initialization list is a symbol whose value is a list of *initializations*, put on by various programs, all to be performed when a certain event (such as a cold boot) happens. When the event occurs, the system function in charge of handling the event (si:lisp-reinitialize, for cold boot) executes all the initializations on the appropriate list, in the order they are present on the list.

Each initialization has a name, a form to be evaluated, a flag saying whether the form has yet been evaluated, and the source file of the initialization, if any. The name is a string or a symbol and lies in the car of an initialization; thus assoc may be used on initialization lists to find particular initializations.

System and user files place initializations on initialization lists using the function add-initialization. The name of the initialization is specified so that the system can distinguish between adding a new initialization and repeating or changing the definition of an initialization already known: if there is already an initialization with the specified name, this is a new definition of the same initialization. One can specify that the initialization be executed immediately if it is new but not if it is repeated.

User programs are free to create their own initialization lists to be run at their own times.

33.1 System Initialization Lists

There are several initialization lists built into the system. Each one is invoked by the system at a specific time, such as immediately after a cold boot, or during disk-save. A user program can put initializations on these lists to cause actions to be taken at those times as the program needs. This avoids the need to modify system functions such as lisp-reinitialize or disk-save in order to make them interact properly with the user program.

The system initialization lists are generally identified by keywords rather than by their actual names. We name them here by their keywords. In each case, the actual initialization list symbol is in the si package, and its name is the conventional keyword followed by '-initialization-list'. Thus, for :cold, there is si:cold-initialization-list. This is just a convention.

Unless otherwise specified, an initialization added to a system list is not run when it is added, only when the appropriate event happens. A few system lists are exceptions and also run each initialization when it is added. Such exceptions are noted explicitly.

The :once initialization list is used for initializations that need to be done only once when the subsystem is loaded and must never be done again. For example, there are some databases that need to be initialized the first time the subsystem is loaded, but should not be reinitialized every time a new version of the software is loaded into a currently running system. This list is for that purpose. When a new initialization is added to this list, it is executed immediately; but when an initialization is redefined, it is not executed again.

The :cold initialization list is used for things that must be run once at cold-boot time. The initializations on this list are run after the ones on :system but before the ones on the :warm list.

The :warm initialization list is used for things which must be run every time the machine is booted, including warm boots. The function that prints the greeting, for example, is on this list. For cold boots, the :cold initializations are done before the :warm ones.

The :system initialization list is like the :warm list but its initializations are run *before* those of the :cold list. These are generally very fundamental system initializations that must be done before the :cold or :warm initializations can work. Initializing the process and window systems, the file system, and the Chaosnet NCP falls in this category. By default, a new initialization added to this list is run immediately also. In general, the system list should not be touched by user subsystems, though there may be cases when it is necessary to do so.

The :before-cold initialization list is used for things to be run by disk-save. Thus they happen essentially at cold boot time, but only once when the world is saved, not each time it is started up.

The :site initialization list is run every time a new site table and host table are loaded by update-site-configuration-info. By default, adding an initialization to this list runs the initialization immediately, even if the initialization is not new.

The :site-option initialization list is run every time the site options may have changed; that is, when a new site tables are loaded or after a cold boot (to see the per-machine options of the machine being booted on). By default, adding an initialization to this list runs the initialization immediately, even if the initialization is not new.

The :full-gc initialization list is run by the function si:full-gc just before garbage collecting. Initializations might be put on this list to discard pointers to bulky objects, or to turn copy lists into cdr-coded form so that they will remain permanently localized.

The :after-flip initialization list is run after every garbage collection flip, at the beginning of scavenging. These initializations can force various objects to be copied into new space near each other simply by referencing them all consecutively.

The :after-full-gc initialization list is run by the function si:full-gc just after a flip is done, but before scavenging.

The :login and :logout lists are run by the **login** and **logout** functions (see page 801) respectively. Note that **disk-save** calls **logout**. Also note that often people don't call **logout**; they just cold-boot the machine.

33.2 Programming Initializations

add-initialization *name form* &optional *list-of-keywords initialization-list-name*

Adds an initialization called *name* with the form *form* to the initialization list specified either by *initialization-list-name* or by keyword. If the initialization list already contains an initialization called *name*, it is redefined to execute *form*.

initialization-list-name, if specified, is a symbol that has as its value the initialization list. If it is void, it is initialized (!) to **nil**, and is given a **si:initialization-list** property of **t**. If a keyword specifies an initialization list, *initialization-list-name* is ignored and should not be specified.

The keywords allowed in *list-of-keywords* are of two kinds. Most specify the initialization list to use; a list of such keywords makes up most of the previous section. Aside from them, four other keywords are allowed, which specify when to evaluate *form*. They are called the *when-keywords*. Here is what they mean:

:normal	Only place the form on the list. Do not evaluate it until the time comes to do this kind of initialization. This is the default unless **:system**, **:once**, :site or :site-option is specified.
:first	Evaluate the form now if it is not flagged as having been evaluated before. This is the default if :system or :once is specified.
:now	Evaluate the form now unconditionally as well as adding it to the list.
:redo	Do not evaluate the form now, but set the flag to **nil** even if the initialization is already in the list and flagged **t**.

Actually, the keywords are compared with **string-equal** and may be in any package. If both kinds of keywords are used, the list keyword should come *before* the when-keyword in *list-of-keywords*; otherwise the list keyword may override the when-keyword.

The **add-initialization** function keeps each list ordered so that initializations added first are at the front of the list. Therefore, by controlling the order of execution of the additions, you can control explicit dependencies on order of initialization. Typically, the order of additions is controlled by the loading order of files. The system list is the most critically ordered of the predefined lists.

The **add-initialization** keywords that specify an initialization list are defined by a variable; you can add new keywords to it.

34.1 Getting and Setting the Time

get-decoded-time
time:get-time

Gets the current time, in decoded form. Return seconds, minutes, hours, date, month, year, day-of-the-week, and daylight-savings-time-p, with the same meanings as decode-universal-time (see page 782). If the current time is not known, nil is returned.

The name time:get-time is obsolete.

get-universal-time

Returns the current time in Universal Time form.

time:set-local-time &optional *new-time*

Sets the local time to *new-time*. If *new-time* is supplied, it must be either a universal time or a suitable argument to time:parse-universal-time (see page 781). If it is not supplied, or if there is an error parsing the argument, you are prompted for the new time. Note that you will not normally need to call this function; it is useful mainly when the timebase gets screwed up for one reason or another.

34.1.1 Elapsed Time in 60ths of a Second

The following functions deal with a different kind of time. These are not calendrical date/times, but simply elapsed time in 60ths of a second. These times are used for many internal purposes where the idea is to measure a small interval accurately, not to depend on the time of day or day of month.

time

Returns a number that increases by 1 every 60th of a second. The value wraps around roughly once a day. Use the time-lessp and time-difference functions to avoid getting in trouble due to the wrap-around. time is completely incompatible with the Maclisp function of the same name.

Note that time with an argument measures the length of time required to evaluate a form. See page 794.

get-internal-run-time
get-internal-real-time

Returns the total time in 60ths of a second since the last boot. This value does not wrap around. Eventually it becomes a bignum. The Lisp Machine does not distinguish between run time and real time.

internal-time-units-per-second *Constant*

According to Common Lisp, this is the ratio between a second and the time unit used by values of get-internal-real-time. On the Lisp Machine, the value is 60. The value may be different in other Common Lisp implementations.

time-lessp *time1 time2*

t if *time1* is earlier than *time2*, compensating for wrap-around, otherwise **nil**.

time-difference *time1 time2*

Assuming *time1* is later than *time2*, returns the number of 60ths of a second difference between them, compensating for wrap-around.

time-increment *time interval*

Increments *time* by *interval*, wrapping around if appropriate.

34.1.2 Elapsed Time in Microseconds

time:microsecond-time

Returns the value of the microsecond timer, as a bignum. The values returned by this function wrap around back to zero about once per hour.

time:fixnum-microsecond-time

Returns as a fixnum the value of the low 23 bits of the microsecond timer. This is like time:microsecond-time, with the advantage that it returns a value in the same format as the time function, except in microseconds rather than 60ths of a second. This means that you can compare fixnum-microsecond-times with time-lessp and time-difference. time:fixnum-microsecond-time is also a bit faster, but has the disadvantage that since you only see the low bits of the clock, the value can wrap around more quickly (about every eight seconds). Note that the Lisp Machine garbage collector is so designed that the bignums produced by time:microsecond-time are garbage-collected quickly and efficiently, so the overhead for creating the bignums is really not high.

34.2 Printing Dates and Times

The functions in this section create printed representations of times and dates in various formats and send the characters to a stream. To any of these functions, you may pass nil as the *stream* parameter and the function will return a string containing the printed representation of the time, instead of printing the characters to any stream.

The three functions time:print-time, time:print-universal-time, time:print-brief-universal-time and time:print-current-time accept an argument called *date-print-mode*, whose purpose is to control how the date is printed. It always defaults to the value of time:*default-date-print-mode*. Possible values include:

:dd//mm//yy	Print the date as in '3/16/53'.
:mm//dd//yy	Print as in '16/3/53'.
:dd-mm-yy	Print as in '16-3-53'.
:dd-mmm-yy	Print as in '16-Mar-53'.
:|dd mmm yy|	Print as in '16 Mar 53'.
:ddmmmyy	Print as in '16Mar53'.

:yymmdd	Print as in '530316'.
:yymmmdd	Print as in '53Mar16'.

time:print-current-time &optional (*stream* *standard-output*)
Prints the current time, formatted as in 11/25/80 14:50:02, to the specified stream. The date portion may be printed differently according to the argument *date-print-mode*.

time:print-time *seconds minutes hours date month year* &optional (*stream* *standard-output*) *date-print-mode*
Prints the specified time, formatted as in 11/25/80 14:50:02, to the specified stream. The date portion may be printed differently according to the argument *date-print-mode*.

time:print-universal-time *universal-time* &optional (*stream* *standard-output*) (*timezone* time:*timezone*) *date-print-mode*
Prints the specified time, formatted as in 11/25/80 14:50:02, to the specified stream. The date portion may be printed differently according to the argument *date-print-mode*.

time:print-brief-universal-time *universal-time* &optional (*stream* *standard-output*) *reference-time date-print-mode*
This is like time:print-universal-time except that it omits seconds and only prints those parts of *universal-time* that differ from *reference-time*, a universal time that defaults to the current time. Thus the output is in one of the following three forms:

```
02:59             ;the same day
3/4 14:01         ;a different day in the same year
8/17/74 15:30     ;a different year
```

The date portion may be printed differently according to the argument *date-print-mode*.

time:*default-date-print-mode* *Variable*
Holds the default for the *date-print-mode* argument to each of the functions above. Initially the value here is :mm//dd//yy.

time:print-current-date &optional (*stream* *standard-output*)
Prints the current time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-date *seconds minutes hours date month year day-of-the-week* &optional (*stream* *standard-output*)
Prints the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-universal-date *universal-time* &optional (*stream* *standard-output*) (*timezone* time:*timezone*)
Prints the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

34.3 Reading Dates and Times

These functions accept most reasonable printed representations of date and time and convert them to the standard internal forms. The following are representative formats that are accepted by the parser. Note that slashes are escaped with additional slashes, as is necessary if these strings are input in traditional syntax.

```
"March 15, 1960"      "3//15//60"    "3//15//1960"
"15 March 1960"       "15//3//60"    "15//3//1960"
"March-15-60"         "3-15-60"      "3-15-1960"
"15-March-60"         "15-3-60"      "15-3-1960"
"15-Mar-60"           "3-15"         "15 March 60"
"Fifteen March 60"    "The Fifteenth of March, 1960;"
"Friday, March 15, 1980"

"1130."     "11:30"      "11:30:17"  "11:30 pm"
"11:30 AM"  "1130"       "113000"
"11.30"     "11.30.00"   "11.3"      "11 pm"

"12 noon"   "midnight"   "m"    "6:00 gmt"   "3:00 pdt"
```

any date format may be used with any time format

```
"One minute after March 3, 1960"
```
meaning one minute after midnight
```
"Two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"

"Now"     "Today"     "Yesterday"    "five days ago"
"two days after tomorrow"    "the day after tomorrow"
"one day before yesterday"   "BOB@OZ's birthday"
```

time:parse *string* &optional (*start* 0) (*end* nil) (*futurep* t) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* t)

Interpret *string* as a date and/or time, and return seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and relative-p. *start* and *end* delimit a substring of the string; if *end* is nil, the end of the string is used. *must-have-time* means that *string* must not be empty. *date-must-have-year* means that a year must be explicitly specified. *time-must-have-second* means that the second must be specified. *day-must-be-valid* means that if a day of the week is given, then it must actually be the day that corresponds to the date. *base-time* provides the defaults for unspecified components; if it is nil, the current time is used. *futurep* means that the time should be interpreted as being in the future; for example, if the base time is 5:00 and the string refers to the time 3:00, that means the next day if *futurep* is non-nil, but it means two hours ago if *futurep* is nil. The *relative-p* returned value is t if the string included a relative part, such as 'one minute after' or 'two days before' or 'tomorrow' or 'now'; otherwise, it is nil.

If the input is not valid, the error condition sys:parse-error is signaled (see page 505).

time:parse-universal-time *string* &optional (*start* 0) (*end* nil) (*futurep* t) *base-time* *must-have-time* *date-must-have-year* *time-must-have-second* (*day-must-be-valid* t)

This is the same as time:parse except that it returns two values: an integer, representing the time in Universal Time, and the *relative-p* value.

34.4 Reading and Printing Time Intervals

In addition to the functions for reading and printing instants of time, there are other functions specifically for printing time intervals. A time interval is either a number (measured in seconds) or nil, meaning 'never'. The printed representations used look like '3 minutes 23 seconds' for actual intervals, or 'Never' for nil (some other synonyms and abbreviations for 'never' are accepted as input).

time:print-interval-or-never *interval* &optional (*stream* *standard-output*)

interval should be a non-negative fixnum or nil. Its printed representation as a time interval is written onto *stream*.

time:parse-interval-or-never *string* &optional *start* *end*

Converts *string*, a printed representation for a time interval, into a number or *nil*. *start* and *end* may be used to specify a portion of *string* to be used; the default is to use all of *string*. It is an error if the contents of string do not look like a reasonable time interval. Here are some examples of acceptable strings:

```
"4 seconds"         "4 secs"           "4 s"
"5 mins 23 secs"    "5 m 23 s"         "23 SECONDS 5 M"
            "3 yrs 1 week 1 hr 2 mins 1 sec"
"never"             "not ever"         "no"             ""
```

Note that several abbreviations are understood, the components may be in any order, and case (upper versus lower) is ignored. Also, "months" are not recognized, since various months have different lengths and there is no way to know which month is being spoken of. This function always accepts anything that was produced by time:print-interval-or-never; furthermore, it returns exactly the same fixnum (or nil) that was printed.

time:read-interval-or-never &optional (*stream* *standard-input*)

Reads a line of input from *stream* (using readline) and then calls time:parse-interval-or-never on the resulting string.

34.5 Time Conversions

decode-universal-time *universal-time* &optional (*timezone* time:*timezone*)

Converts *universal-time* into its decoded representation. The following values are returned: seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and the timezone used. *daylight-savings-time-p* tells you whether or not daylight savings time is in effect; if so, the value of *hour* has been adjusted accordingly. You can specify *timezone* explicitly if you want to know the equivalent representation for this time in other parts of the world.

encode-universal-time *seconds minutes hours date month year* &optional *timezone*

Converts the decoded time into Universal Time format, and return the Universal Time as an integer. If you don't specify *timezone*, it defaults to the current timezone adjusted for daylight savings time; if you provide it explicitly, it is not adjusted for daylight savings time. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:*timezone* *Variable*

The value of time:*timezone* is the time zone in which this Lisp Machine resides, expressed in terms of the number of hours west of GMT this time zone is. This value does not change to reflect daylight savings time; it tells you about standard time in your part of the world.

34.6 Internal Functions

These functions provide support for those listed above. Some user programs may need to call them directly, so they are documented here.

time:initialize-timebase

Initializes the timebase by querying Chaosnet hosts to find out the current time. This is called automatically during system initialization. You may want to call it yourself to correct the time if it appears to be inaccurate or downright wrong. See also **time:set-local-time**, page 777.

time:daylight-savings-time-p *hours date month year*

Returns t if daylight savings time is in effect for the specified hour; otherwise, return **nil**. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:daylight-savings-p

Returns t if daylight savings time is currently in effect; otherwise, returns **nil**.

time:month-length *month year*

Returns the number of days in the specified *month*; you must supply a *year* in case the month is February (which has a different length during leap years). If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:leap-year-p *year*

Returns t if *year* is a leap year; otherwise return nil. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:verify-date *date month year day-of-the-week*

If the day of the week of the date specified by *date*, *month*, and *year* is the same as *day-of-the-week*, returns nil; otherwise, returns a string that contains a suitable error message. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:day-of-the-week-string *day-of-the-week* &optional (*mode* :long)

Returns a string representing the day of the week. As usual, 0 means Monday, 1 means Tuesday, and so on. Possible values of *mode* are:

:long	Returns the full English name, such as "Monday", "Tuesday", etc. This is the default.
:short	Returns a three-letter abbreviation, such as "Mon", "Tue", etc.
:medium	Returns a longer abbreviation, such as "Tues" and "Thurs".
:french	Returns the French name, such as "Lundi", "Mardi", etc.
:german	Returns the German name, such as "Montag", "Dienstag", etc.
:italian	Returns the Italian name, such as "Lunedi", "Martedi", etc.

time:month-string *month* &optional (*mode* :long)

Returns a string representing the month of the year. As usual, 1 means January, 2 means February, etc. Possible values of *mode* are:

:long	Returns the full English name, such as "January", "February", etc. This is the default.
:short	Returns a three-letter abbreviation, such as "Jan", "Feb", etc.
:medium	Returns a longer abbreviation, such as "Sept", "Novem", and "Decem".
:roman	Returns the Roman numeral for *month* (this convention is used in Europe).
:french	Returns the French name, such as "Janvier", "Fevrier", etc.
:german	Returns the German name, such as "Januar", "Februar", etc.
:italian	Returns the Italian name, such as "Gennaio", "Febbraio", etc.

time:timezone-string &optional (*timezone* time:*timezone*)
(*daylight-savings-p* (time:daylight-savings-p))

Return the three-letter abbreviation for this time zone. For example, if *timezone* is 5, then either "EST" (Eastern Standard Time) or "CDT" (Central Daylight Time) is used, depending on *daylight-savings-p*.

35. Miscellaneous Useful Functions

This chapter describes a number of functions that don't logically fit in anywhere else. Most of these functions are not normally used in programs, but are "commands", i.e. things that you type directly at Lisp.

35.1 Documentation

documentation *name doc-type*

Returns the documentation string of *name* in the role *doc-type*. *doc-type* should be a symbol, but only its print-name matters. **function** as *doc-type* requests the documentation of *name* as a function, **variable** as *doc-type* requests the documentation of *name* as a variable, and so on.

When *doc-type* is **function**, *name* can be any function spec, and the documentation string of its function definition is returned. Otherwise, *name* must be a symbol, and *doc-type* may be anything. However, only these values of *doc-type* are standardly used:

variable	Documentation of *name* as a special variable. Such documentation is recorded automatically by **defvar**, **defconst**, **defconstant**, **defparameter** (page 33).
type	Documentation of *name* as a type for **typep**. Recorded automatically when a documentation string is given in a **deftype** form (page 19).
structure	Documentation of *name* as a **defstruct** type. Recorded automatically by a **defstruct** for *name* (chapter 20, page 372).
setf	Documentation on what it means to **setf** a form that starts with *name*. Recorded when there is a documentation string in a **defsetf** of *name* (page 345).
flavor	Documentation of the flavor named *name*. Put on by the **:documentation** option in a **defflavor** for *name* (page 414).
resource	Documentation of the resource named *name*. Put on when there is a documentation string in a **defresource** of *name* (page 124).
signal	Documentation for *name* as a signal name. Put on when there is a documentation string in a **defsignal** or **defsignal-explicit** for *name* (page 714).

Documentation strings for any *doc-type* can be added to *name* by doing **(setf (documentation** *name doc-type*) *string*).

The command **Control-Shift-D** in Zmacs and the rubout handler, used within a call to a function, prints the documentation of that function. **Control-Shift-V**, within a symbol, prints the documentation of that symbol as a variable.

35.2 Hardcopy

The hardcopy functions allow you to specify the printer to use on each call. The default is set up by the site files for your site, but can be overridden for a particular machine in the LMLOCS file or by a user in his INIT file. Any kind of printer can be used, no matter how it is actually driven, if it is hooked into the software properly as described below.

A *printer-type* is a keyword that has appropriate properties; a *printer* is either a printer-type or a list starting with one. The rest of the list can specify *which* printer of that type you want to use (perhaps with a host name or filename).

The printer types defined by the system are:

:dover	This printer type is used by itself as a printer, and refers to the Dover at MIT.
:xgp	This printer type indicates a printer that is accessed by writing spool files in MIT XGP format. A printer would be specified as a list, (:xgp *filename*), specifying where to write the spool file.
:press-file	This printer type is used in a list together with a file name, as in (:press-file "OZ:<RMS>FOO.PRESS"). Something is "printed" on such a printer by being converted to a press file and written under that name.

hardcopy-file *filename* &rest *options*

Print the file *filename* in hard copy on the specified printer or the default printer. *options* is a list of keyword argument names and values. There are only two keywords that are always meaningful: :format and :printer. Everything else is up to the individual printer to interpret. The list here is only a paradigm or suggestion.

:printer	The value is the printer to use. The default is the value of **si:*default-printer***.
:format	The value is a keyword that specifies the format of file to be parsed. The standard possibilities are :text (an ordinary file of text), :xgp (a file of the sort once used by the XGP at MIT), :press (a Xerox-style press file) and :suds-plot (a file produced by the Stanford drawing program). However, each kind of printer may define its own format keywords.
:font	
:font-list	The value of *font* is the name of a font to print the file in (a string). Alternatively, you can give :font-list and specify a list of such font names, for use if the file contains font-change commands. The interpretation of a font name is dependent on the printer being used. There is no necessary relation to Lisp machine display fonts. However, printers are encouraged to use, by default, fonts that are similar in appearance to the Lisp machine fonts listed in the file's attribute list, if it is a text file.
:heading-font	The value is the name of the font for use in page headers, if there are any.
:vsp	The value is the extra spacing to use between lines, in over and beyond the height of the fonts.

:page-headings
If the value is non-nil, a heading is added to each page.

:copies The value is the number of copies to print.

:spool If the printer provides optional spooling, this argument says whether to spool (default is nil). Some printers may intrinsically always spool; others may have no way to spool.

set-printer-default-option *printer-type option value*

Sets a default for *option* for printers of type *printer-type*. Any use of the hardcopy functions with a printer of that type and no value specified for *option* will use the value *value*. For example,

```
    (set-printer-default-option :dover :spool t)
```

causes output to Dover printers to be spooled unless the :spool option is explicitly specified with value nil.

Currently defaultable options are :font, :font-list, :heading-font, :page-headings, :vsp, :copies, and :spool.

hardcopy-stream *stream* &rest *options*

Like hardcopy-file but uses the text read from *stream* rather than opening a file. The :format option is not allowed (since implementing it requires the ability to open the file with unusual open options).

hardcopy-bit-array *array left top right bottom* &rest *options*

Print all or part of the bit-array *array* on the specified or default printer. *options* is a list of keyword argument names and values; the only standard option is :printer, which specifies the printer to use. The default printer is si:*default-bit-array-printer*, or, if that is nil, si:*default-printer.

left, *top*, *right* and *bottom* specify the subrectangle of the array to be printed. All four numbers measure from the top left corner (which is element 0, 0).

hardcopy-status &optional *printer* (*stream* *standard-output*)

Prints the status of *printer*, or the default printer. This should include if possible such things as whether the printer has paper and what is in the queue.

si:*default-printer* *Variable*

This is the default printer. It is set from the :default-printer site option.

si:*default-bit-array-printer* *Variable*

If non-nil, this is the default printer for printing bit arrays, overriding si:*default-printer*. A separate default is provided for bit arrays since some printers that can print files cannot print bit arrays. This variable is set initially from the :default-bit-array-printer site option.

Defining a printer type:

A printer type is any keyword that has suitable functions on the appropriate properties.

To be used with the function **hardcopy-file**, the printer type must have a **si:print-file** property. To be used with **hardcopy-stream**, the printer type must have a **si:print-stream** property. **hardcopy-bit-array** uses the **si:print-bit-array** property. **hardcopy-status** uses the **si:print-status** property. (The hardcopy functions' names are not themselves used simply to avoid using a symbol in the global package as a property name of a symbol that might be in the global package as well).

Each property, to be used, should be a function whose first argument will be the printer and whose remaining arguments will fit the same pattern as those of the hardcopy function the user called. (They will not necessarily be the same arguments, as some additional keywords may be added to the list of keyword arguments; but they will fit the same description.)

For example,

```
    (hardcopy-file "foo" :printer '(:press-file "bar.press"))
```

results in the execution of

```
    (funcall (get :press-file 'si:print-file)
             '(:press-file "bar.press")
             "foo" :printer '(:press-file "bar.press"))
```

A printer type need not support operations that make no sense on it. For example, there is no **si:print-status** property on **:press-file**.

35.3 Metering

The metering system is a way of finding out what parts of your program use up the most time. When you run your program with metering, every function call and return is recorded, together with the time at which it took place. Page faults are also recorded. Afterward, the metering system analyzes the records and tells you how much time was spent executing withain each function. Because the records are stored in the disk partition called **METR**, there is room for a lot of data.

Before you meter a program, you must enable metering in some or all stack groups. **meter:enable** is used for this. Then you evaluate one or more forms with metering, perhaps by using **meter:test** or **meter:run**. Finally, you use **meter:analyze** to summarize and print the metering data.

There are two parameters that control whether metering data are recorded. First of all, the variable **sys:%meter-microcode-enables** contains bits that enable recording of various kinds of events. Secondly, each stack group has a flag that controls whether events are recorded while running in that stack group.

sys:%meter-microcode-enables *Variable*

Enables recording of metering data. Each bit controls recording of one kind of event.

1	This bit enables recording of page faults.
2	This bit enables recording of consing.
4	This bit enables recording of function entry and exit.
8	This bit enables recording of stack group switching.

The value is normally zero, which turns off all recording.

These are the functions used to control which stack groups do metering:

meter:enable &rest *things*

Enables metering in the stack groups specified by *things*. Each thing in *things* may be a stack group, a process (which specifies the process's stack group), or a window (which specifies the window's process's stack group). t is also allowed. It enables metering in all stack groups.

meter:disable &rest *things*

Disables metering in the stack groups specified by *things*. The arguments allowed are the same as for meter:enable. (meter:disable t) turns off (meter:enable t), but does not disable stack groups enabled individually.. (meter:disable) disables all stack groups no matter how you specified to enable them.

meter:metered-objects *Variable*

This is a list of all the *things* you have enabled with meter:enable and not disabled.

These are the functions to evaluate forms with metering:

meter:run *forms*

Clears out the metering data and evaluates the *forms* with sys:%meter-microcode-enables bound to 14 octal (record function entry and exit, and stack group switching). Any of the evaluation that takes place in enabled stack groups will record metering data.

meter:test *form* (*enables* #o14)

Clears out the metering data, enables metering for the current stack group only, and evaluates *form* with sys:%meter-microcode-enables bound to *enables*.

This is how you print the results:

meter:analyze &key *analyzer stream file buffer return info* &allow-other-keys

Analyzes the data recorded by metering. *analyzer* is a keyword specifies a kind of analysis. :tree is the default. Another useful alternative is :list-events. Particular analyzers handle other keyword arguments in addition to those listed above.

The output is printed on *stream*, written to a file named *file*, or put in an editor buffer named *buffer* (at most one of these three arguments should be specified). The default is to print on *standard-output*.

Analyzing the metering data involves creating a large intermediate data base. Normally this is created afresh each time **meter:analyze** is called. If you specify a non-**nil** value for *return*, the intermediate data structure is returned by **meter:analyze**, and can be passed in on another call as the *info* argument. This can save time. But you can only do this if you use the same *analyzer* each time, as different analyzers use different termporary data structures.

The default analyzer **:tree** prints out the amount of run time and real time spent executing each function that was called. The real time includes time spend waiting and time spent writing metering data to disk; for computational tasks, the latter makes the real time less useful than the run time. **:tree** handles these additional keyword arguments to **meter:analyze**:

:find-callers	The argument for this keyword is a function spec or a list of function specs. A list of who called the specified functions, and how often, is printed instead of the usual output.
:stack-group	The argument is a stack group or a list of them; only the activities in those stack groups are printed.
:sort-function	The argument is the name of a suitable sorting function that is used to sort the items for the various functions that were called. Sorting functions provided include **meter:max-page-faults**, **meter:max-calls**, **meter:max-run-time** (the default), **meter:max-real-time**, and **meter:max-run-time-per-call**.
:summarize	The argument is a function spec or a list of function specs; only those functions' statistics are printed.
:inclusive	If this is non-**nil**, the times for each function include the time spent in executing subroutines called from the function. Note: if a function is called recursively, the time spent in the inner call(s) is counted twice (or more).

The analyzer **:list-events** prints out one line about each event recorded. The line contains the run time and real time (in microseconds), the running count of page faults, the stack group name, the function that was running, the stack depth, the type of event, and a piece of data. For example:

```
    0   0    0 ZMACS-WINDOWS  METER:TEST   202 CALL SI:EVAL
  115  43    0 ZMACS-WINDOWS  METER:TEST   202 RET  SI:EVAL
  180  87    0 ZMACS-WINDOWS  METER:TEST   202 RET  CATCH

 real run   pf  stack-group    function   stack event data
 time time                                level type
```

:list-events is often useful with recording of page faults (**sys:%meter-microcode-enables** set to 1).

meter:reset

Clears out all metering data.

Because metering records pointers to Lisp objects in a disk partition which is not part of the Lisp address space, garbage collection is inhibited (by arresting the gc process) when you turn on metering.

meter:resume-gc-process

Allows garbage collection to continue (if it is already turned on) by unarresting it.

35.4 Poking Around in the Lisp World

who-calls *x* &optional *package* (*inheritors* t) (*inherited* t)

x must be a symbol or a list of symbols. who-calls tries to find all of the functions in the Lisp world that call *x* as a function, use *x* as a variable, or use *x* as a constant. (Constants which are lists containing *x* are not found.) It tries to find all of the functions by searching all of the function cells of all of the symbols in *package* and packages that inherit from *package* (unless *inheritors* is nil) and packages *package* inherits from (unless *inherited* is nil). *package* defaults to the global package, which means that all normal packages are checked.

If who-calls encounters an interpreted function definition, it simply tells you if *x* appears anywhere in the interpreted code. who-calls is smarter about compiled code, since it has been nicely predigested by the compiler. Macros expanded in the compilation of the code can be found because they are recorded in the caller's debugging info alist, even though they are not actually referred to by the compiled code.

If *x* is a list of symbols, who-calls does them all simultaneously, which is faster than doing them one at a time.

who-uses is an obsolete name for who-calls.

The editor has a command, Meta-X List Callers, which is similar to who-calls.

The symbol :unbound-function is treated specially by who-calls. (who-calls :unbound-function) searches all the compiled code for any calls through a symbol that is not currently defined as a function. This is useful for finding errors such as functions you misspelled the names of or forgot to write.

who-calls prints one line of information for each caller it finds. It also returns a list of the names of all the callers.

what-files-call *x* &optional *package* (*inheritors* t) (*inherited* t)

Similar to who-calls but returns a list of the pathnames of all the files that contain functions that who-calls would have printed out. This is useful if you need to recompile and/or edit all of those files.

apropos *substring* &optional *package* &key (*inheritors* t) *inherited dont-print predicate boundp fboundp*

(apropos *substring*) tries to find all symbols whose print-names contain *substring* as a substring. Whenever it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so and prints the names of the arguments (if any) to the function.

If *predicate* is non-nil, it should be a function; only symbols on which the function returns non-nil are counted. In addition, *fboundp* non-nil means only symbols with function definitions are considered, and *boundp* non-nil means that only symbols with values are considered.

apropos looks for symbols on *package*, and all packages that use *package* (unless *inheritors* is nil). If *inherited* is non-nil, all packages used by *package* are searched as well. *package* can be a package or a symbol or string naming a package. It can also be a list of packages, symbols and strings; all of the packages thus specified are searched. *package* defaults to a list of all packages except invisible ones.

apropos returns a list of all the symbols it finds. If *dont-print* is non-nil, that is all it does.

sub-apropos *substring starting-list* &key *predicate dont-print*

Finds all symbols in *starting-list* whose names contain *substring*, and that satisfy *predicate*. If *predicate* is nil, the substring is the only condition. The symbols are printed if *dont-print* is nil. A list of the symbols found is returned, in any case.

This function is most useful when applied to the value of *, after apropos has returned a long list.

where-is *pname* &optional *package*

Prints the names of all packages that contain a symbol with the print-name *pname*. If *pname* is a string it gets upper-cased. The package *package* and all packages that inherit from it are searched. *package* can be a package or the name of a package, or a list of packages and names. It defaults to a list of all packages except invisible ones. where-is returns a list of all the symbols it finds.

describe *x*

describe tries to tell you all of the interesting information about any object *x* (except for array contents). describe knows about arrays, symbols, floats, packages, stack groups, closures, and FEFs, and prints out the attributes of each in human-readable form. Sometimes objects found inside *x* are described also; such recursive descriptions are indented appropriately. For instance, describe of a symbol also describes the symbol's value, its definition, and each of its properties. describe of a float (full-size or short) shows you its internal representation in a way that is useful for tracking down roundoff errors and the like.

If *x* is a named-structure, describe invokes the :describe operation to print the description, if that is supported. To understand this, you should read the section on named structures (see page 390). If the :describe operation is not supported, describe looks on the named-structure symbol for information that might have been left by **defstruct**; this information would tell it what the symbolic names for the entries in the structure are, and **describe** knows how to use the names to print out what each field's name and contents is.

describe of an instance always invokes the **:describe** operation. All flavors support it, since **si:vanilla-flavor** defines a method for it.

describe always returns its argument, in case you want to do something else to it.

inspect *x*

A window-oriented version of **describe**. See the window system documentation for details, or try it and type **Help**.

disassemble *function*

Prints out a human-readable version of the macro-instructions in *function*. *function* should be a FEF, or a function spec whose definition is a FEF. The macro-code instruction set is explained in chapter 31, page 752.

The **grindef** function (see page 528) may be used to display the definition of a non-compiled function.

room &rest *areas*

Prints a summary of memory usage.

The first line of output tells you the amount of physical memory on the machine, the amount of available virtual memory not yet filled with data (that is, the portion of the available virtual memory that has not yet been allocated to any region of any area), and the amount of wired physical memory (i.e. memory not available for paging).

Following lines tell you how much room is left in some areas. For each area it tells you about, it prints out the name of the area, the number of regions that currently make up the area, the current size of the area in kilowords, and the amount of the area that has been allocated, also in kilowords. If the area cannot grow, the percentage that is free is displayed.

(room) tells you about those areas that are in the list that is the value of the variable **room**. These are the most interesting ones.

(room *area1 area2*...**)** tells you about those areas, which can be either the names or the numbers.

(room t) tells you about all the areas.

(room nil) does not tell you about any areas; it only prints the first line of output.

room *Variable*

The value of **room** is a list of area names and/or area numbers, denoting the areas that the function **room** should describe if given no arguments. Its initial value is:

```
(working-storage-area macro-compiled-program)
```

35.5 Utility Programs

ed &optional *x*

ed is the main function for getting into the editor, Zmacs. The commands of Zmacs are very similar to those of Emacs.

(ed) or (ed nil) simply enters the editor, leaving you in the same buffer as the last time you were in the editor. It has the same effect as typing System E.

(ed t) puts you in a fresh buffer with a generated name (like BUFFER-4).

(ed *pathname*) edits that file. *pathname* may be an actual pathname or a string.

(ed 'foo) tries hard to edit the definition of the foo function. It can find a buffer or file containing the source code for foo and position the cursor at the beginning of the code. In general, foo can be any function-spec (see section 11.2, page 223).

(ed 'zwei:reload) reinitializes the editor. It forgets about all existing buffers, so use this only as a last resort.

zwei:save-all-files

This function is useful in emergencies in which you have modified material in Zmacs buffers that needs to be saved, but the editor is partially broken. This function does what the editor's Save All Files command does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

dired &optional *pathname*

Puts up a window and edits the directory named by *pathname*, which defaults to the last file opened. While editing a directory you may view, edit, compare, hardcopy, and delete the files and subdirectories it contains. While in the directory editor type the **Help** key for further information.

mail &optional *user text call-editor-anyway*

Sends the string *text* as mail to *user*. *user* should also be a string, of the form "*username*@*hostname*". Multiple recipients separated by commas are also allowed.

If you do not provide two arguments, mail puts up an editor window in which you may compose the mail. Type the End key to send the mail and return from the mail function.

The window is also used if *call-editor-anyway* is non-nil.

bug &optional *topic text call-editor-anyway*

Reports a bug. *topic* is the name of the faulty program (a symbol or a string). It defaults to **lispm** (the Lisp Machine system itself). *text* is a string which contains the information to report. If you do not provide two arguments, or if *call-editor-anyway* is non-**nil**, a window is put up for you to compose the mail.

bug is like **mail** but includes information about the system version and what machine you are on in the text of the message. This information is important to the maintainers of the faulty program; it aids them in reproducing the bug and in determining whether it is one that is already being worked on or has already been fixed.

print-notifications &optional (*from* 0) *to*

Reprints any notifications that have been received. **from** and **to** are used to restrict which notifications are printed; both count from the most recent notification as number 0. Thus, **(print-notifications 2 8)** prints six notifications after skipping the two most recent.

The difference between notifications and sends is that sends come from other users, while notifications are usually asynchronous messages from the Lisp Machine system itself. However, the default way for the system to inform you about a send is to make a notification! So **print-notifications** *normally* includes all sends as well.

Typing **Terminal 1 N** pops up a window and calls **print-notifications** to print on it.

si:print-disk-error-log

Prints information about the half dozen most recent disk errors (since the last cold boot).

peek &optional *character*

Selects the PEEK utility, which displays various information about the system, periodically updating it. PEEK has several modes, which are entered by typing a single key which is the name of the mode or by clicking on the menu at the top. The initial mode is selected by the argument, *character*. If no argument is given, PEEK starts out by explaining what its modes are.

time *form*

Evaluates *form* and prints the length of time that the evaluation took. The values of *form* are returned.

Note that **time** with no argument is a function to return a time value counting in 60ths of a second; see page 777. This unfortunate collision is a consequence of Common Lisp.

35.6 The Lisp Listen Loop

These functions constitute the Lisp top level read-eval-print loop or *listen loop* and its associated functions.

si:lisp-top-level

This is the first function called in the initial Lisp environment. It calls **lisp-reinitialize**, clears the screen, and calls **si:lisp-top-level1**.

lisp-reinitialize

This function does a wide variety of things, such as resetting the values of various global constants and initializing the error system.

si:lisp-top-level1 **terminal-io**

This is the actual listen loop. Within it, ***terminal-io*** is bound to the argument supplied. This is the stream used for reading and printing if ***standard-input*** and ***standard-output*** are synonyms for ***terminal-io***, as they normally are.

The listen loop reads a form from ***standard-input***, evaluates it, prints the result (with escaping) to ***standard-output***, and repeats indefinitely. If several values are returned by the form all of them are printed. Also the values of *****, **+**, **-**, **//**, **++**, ******, **+++**, ******* and ***values*** are maintained (see below).

break *format-string* &rest *format-args*

Enters a breakpoint loop, which is similar to a Lisp top level loop. *format-string* and the *format-args* are passed to **format** to print a message.

```
        ;Breakpoint message; Resume to continue, Abort to quit.
```

and then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top level loop is that when reading a form, **break** checks for the following special cases: If the **Abort** key is typed, control is returned to the previous break or error-handler, or to top-level if there is none. If the **Resume** key is typed, **break** returns **nil**. If the list (**return** *form*) is typed, **break** evaluates *form* and returns the result, without ever calling the function **return**.

Inside the **break** loop, the streams ***standard-output***, ***standard-input***, and **query-io** are bound to be synonymous to ***terminal-io***; ***terminal-io*** itself is not rebound. Several other internal system variables are bound, and you can add your own symbols to be bound by pushing elements onto the value of the variable **sys:*break-bindings*** (see page 797).

break used to be a special form whose first argument was a string or symbol which was simply printed *without evaluating it*. In order to facilitate conversion, **break** really still is a special form. If the call appears to use the old conventions, it behaves in the old way, but the compiler issues a warning if it sees such code.

prin1 *Variable*

The value of this variable is normally **nil**. If it is non-**nil**, then the read-eval-print loop uses its value instead of the definition of **prin1** to print the values returned by functions. This hook lets you control how things are printed by all read-eval-print loops—the Lisp top level, the **break** function, and any utility programs that include a read-eval-print loop. It does not affect output from programs that call the **prin1** function or any of its relatives such as **print** and **format**; if you want to do that, read about customizing the printer, on section 23.1, page 513. If you set **prin1** to a new function, remember that the read-eval-print loop expects the function to print the value but not to output a **return** character or any other delimiters.

- *Variable*

While a form is being evaluated by a read-eval-print loop, **-** is bound to the form itself.

+ *Variable*

While a form is being evaluated by a read-eval-print loop, **+** is bound to the previous form that was read by the loop.

***** *Variable*
// *Variable*

While a form is being evaluated by a read-eval-print loop, ***** is set to the result printed the last time through the loop. If there were several values printed (because of a multiple-value return), ***** is bound to the first value. **//** is bound to a list of all the values of the previous form.

If evaluation of a form is aborted, ***** and **//** remain set to the results of the last successfully completed form. If evaluation is successful but printing is aborted, ***** and **//** are already set for the following form.

Note that when using Common Lisp syntax you would type just **/**.

++ *Variable*

++ holds the previous value of **+**, that is, the form evaluated two interactions ago.

+++ *Variable*

+++ holds the previous value of **++**.

****** *Variable*
//// *Variable*

Hold the previous values of ***** and **//**, that is, the results of the form evaluated two interactions ago. Only forms whose evaluation is successful cause the values of ***** and **//** to move into ****** and **////**.

Note that when using Common Lisp syntax you would type just **//**.

******* *Variable*
////// *Variable*

Hold the previous values of ** and ////, that is, the results of the form evaluated three interactions ago. Note that when using Common Lisp syntax you would type just ///.

values *Variable*

values holds a list of all lists of values produced by evaluation in this Lisp listener. (car *values*) is nearly equivalent to //, (cadr *values*) to ////, and so on. The difference is that an element is pushed on *values* for each form whose evaluation is started. If evaluation is aborted, the element of *values* is nil.

sys:*break-bindings* *Variable*

When break is called, it binds some special variables under control of the list which is the value of sys:*break-bindings*. Each element of the list is a list of two elements: a variable and a form that is evaluated to produce the value to bind it to. The bindings happen sequentially. Users may push things on this list (adding to the front of it), but should not replace the list wholesale since several of the variable bindings on this list are essential to the operation of break.

lisp-crash-list *Variable*

The value of lisp-crash-list is a list of forms. lisp-reinitialize sequentially evaluates these forms, and then sets lisp-crash-list to nil.

In most cases, the *initialization* facility should be used rather than lisp-crash-list. Refer to chapter 33, page 772.

35.7 The Garbage Collector

gc-on

Turns automatic garbage collection on. Garbage collection will happen when and as needed. Automatic garbage collection is off by default.

Since garbage collection works by copying, you are asked for confirmation if there may not be enough space to complete a garbage collection even if it is started immediately.

gc-off

Turns automatic garbage collection off.

gc-on *Variable*

t when garbage collection is on, nil when it is not. You cannot control garbage collection by setting this variable; it exists so you can examine it. In particular, you can tell if the system found it necessary to turn off garbage collection because it was close to running out of virtual memory.

Normally, automatic garbage collection happens in incremental mode; that is, scavenging happens in parallel with computation. Each consing operation scavenges or copies four words per word consed. In addition, scavenging goes on whenever the machine appears idle.

si:inhibit-idle-scavenging-flag *Variable*
If this is non-nil, scavenging is not done during idle time.

If you are running a noninteractive crunching program, the incremental nature of garbage collection may not be helpful. Then you can make garbage collection more efficient by making it a batch process.

si:gc-reclaim-immediately *Variable*
If this variable is non-nil, automatic garbage collection is done as a batch operation: when the garbage collection process decides that the time has come, it copies all the useful data and discards the old address space, running full blast. (It is still possible to use the machine while this is going on, but it is slow.) More specifically, the garbage collection process scavenges and reclaims oldspace immediately right after a flip happens, using all of the machine's physical memory. This variable is only relevant if you have turned on automatic garbage collection with **(gc-on)**.

A batch garbage collection requires less free space than an incremental one. If there is not enough space to complete an incremental garbage collection, you may be able to win by selecting batch garbage collection instead.

si:gc-reclaim-immediately-if-necessary *Variable*
If this variable is non-nil, then automatic garbage collection is done in batch mode if, when the flip is done, there does not seem to be enough space left to do it incrementally. This variable's value is relevant only if **si:gc-reclaim-immediately** is **nil**.

si:gc-flip-ratio *Variable*
This variable tells the garbage collector what fraction of the data it should expect to have to copy, after each flip. It should be a positive number no larger than one. By default, it is one. But if your program is consing considerable amounts of garbage, a value less than one may be safe. The garbage collector uses this variable to figure how much space it will need to copy all the living data, and therefore indirectly how often garbage collection must be done.

si:gc-flip-minimum-ratio *Variable*
This value is used, when non-nil, to control warnings about having too little space to garbage collect. Its value is a positive number no greater than one, just like that of si:gc-flip-ratio. The difference between the two is that si:gc-flip-ratio controls when garbage collection is *recommended*, whereas si:gc-flip-minimum-ratio controls when the system considers the last possible time to do so. If si:gc-flip-minimum-ratio is nil, si:gc-flip-ratio serves both purposes.

Garbage collection is turned off if it appears to be about to run out of memory. You get a notification if this happens. You also get a notification when you are nearly at the point of not having enough space to guarantee garbage collecting successfully.

In addition to turning on automatic garbage collection, you can also manually request one immediate complete collection with the function **si:full-gc**. The usual reason for doing this is to make a band smaller before saving it. **si:full-gc** also resets all temporary areas (see **si:reset-temporary-area**, page 299).

si:full-gc

Performs a complete garbage collection immediately. This does not turn automatic garbage collection on or off; it performs the garbage collection in the process you call it in. A full gc of the standard system takes about 7 minutes, currently.

si:clean-up-static-area *area-number*

This is a more selective way of causing static areas to be garbage collected once. The argument is the area number of a static area; that particular area will be garbage collected the next time a garbage collection is done (more precisely, it will be copied and discarded after the next flip). If you then call **si:full-gc**, it will happen then.

gc-status

The function **gc-status** prints information related to garbage collection. When scavenging is in progress, it tells you how the task is progressing. While scavenging is not in progress and oldspace does not exist, it prints information about how soon a new flip will be required.

While a garbage collection is not in progress, the output from **gc-status** looks like this:

```
Dynamic (new+copy) space 557,417, Old space 0, Static 3,707,242,
Free space 10,453,032, with 10,055,355 needed for garbage collection
 assuming 100% live data (SI:GC-FLIP-RATIO = 1).
If GC is turned on, a flip will happen in 397,677 words.
Scavenging during cons Off, Idle scavenging On,
Automatic garbage collection Off.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

or

```
Dynamic (new+copy) space 561,395, Old space 0, Static 3,707,242,
Free space 10,453,032, with 10,058,670 needed for garbage collection
 assuming 100% live data (SI:GC-FLIP-RATIO = 1).
A flip will happen in 394,362 words.
Scavenging during cons On, Idle scavenging On,
Automatic garbage collection On.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

The "dynamic space" figure is the amount of garbage collectable space and the "static" figure is the amount of static space used. There is no old space since an old space only exists during garbage collection.

The amount of space needed for garbage collection represents an estimate of how much space user programs will use up while scavenging is in progress. It includes a certain amount of padding. The difference between the free space and that amount is how much consing you can do before a garbage collection will begin (if automatic garbage collection is on).

The amount needed for a garbage collection depends on the value of **si:*gc-reclaim-immediately***; more if it is **nil**.

While a garbage collection is in progress, the output looks like this:

```
Incremental garbage collection now in progress.
Dynamic (new+copy) space 45,137, Old space 972,514, Static 3,707,498,
Between 3,701,440 and 4,629,998 words of scavenging left to do.
Free space 9,289,795 (of which 928,558 might be needed for copying).
Ratio scavenging work/free space = 0.55.
Scavenging during cons On, Idle scavenging On,
Automatic garbage collection On.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

Notice that most of the dynamic space has become old space and new space is small. Not much has been copied since the flip took place. The maximum and minimum estimates for the amount of scavenging are based on different limits for how much of old space may need to be copied; as scavenging progresses, the maximum decreases steadily, but the minimum may increase. The free space is smaller now, but it will get larger when scavenging is finished and old space is freed up. (The total amounts are not the same now because unused parts of regions may not be included in any of the figures.)

si:set-scavenger-ws *number-of-pages*

Incremental scavenging is restricted to a fixed amount of physical memory to reduce its interference with your other activities.

This function specifies the number of pages of memory that incremental garbage collection can use. 256 is a good value for a 256k machine. If the garbage collector gets very poor paging performance, use of this function may fix it.

35.8 Logging In

Logging in tells the Lisp Machine who you are, so that other users can see who is logged in, you can receive messages, and your INIT file can be run. An INIT file is a Lisp program which gets loaded when you log in; it can be used to set up a personalized environment.

When you log out, it should be possible to undo any personalizations you have made so that they do not affect the next user of the machine. Therefore, anything done by an INIT file should be undoable. In order to do this, for every form in the INIT file, a Lisp form to undo its effects should be added to the list that is the value of logout-list. The login-forms construct helps make this easy; see below.

user-id *Variable*

The value of user-id is either the name of the logged in user, as a string, or else an empty string if there is no user logged in. It appears in the who-line.

logout-list *Variable*

The value of logout-list is a list of forms to be evaluated when the user logs out.

login *name* &optional *host inhibit-init-file*
Sets your name (the variable **user-id**) to *name* and logs in a file server on *host*. *host* also becomes your default file host. The default value of *host* depends on which Lisp Machine you use using; it is called the associated machine (see page 815). **login** also runs the :login initialization list (see page 773).

If *host* requires passwords for logging in you are asked for a password. Adding an asterisk at the front of your password enables any special capabilities you may be authorized to use, by calling **fs:enable-capabilities** (page 609).

Unless *inhibit-init-file* is specified as non-nil, **login** loads your init file if it exists. On ITS, your init file is *name* LISPM on your home directory. On TOPS-20 your init file is LISPM.INIT on your directory. On VMS, it is LISPM.INI. On Unix, it is **lispm.init**.

If anyone is logged into the machine already, **login** logs him out before logging in *name*. (See **logout**.) Init files should be written using the **login-forms** construct so that **logout** can undo them. Usually, however, you cold-boot the machine before logging in, to remove any traces of the previous user. **login** returns **t**.

log1 &rest *options*
Like **login** but the arguments are specified differently. *options* is a list of keywords and values; the keywords **:host** and **:init** specify the host to log in on and whether to load the init file if any. Any other keywords are also allowed. **log1** itself ignores them, but the init file can act on them. The purpose of **log1**, as opposed to **login**, is to enable you to specify other keywords for your init file's sake.

si:user-init-options *Variable*
During the execution of the user's init file, inside **log1**, this variable contains the arguments given to **log1**. Options not meaningful to **log1** itself can be specified, so that the init file can find them here and act on them.

logout
First, **logout** evaluates the forms on **logout-list**. Then it sets **user-id** to an empty string and **logout-list** to **nil**. Then it runs the **:logout** initialization list (see page 773), and returns **t**.

login-forms *undoable-forms...* *Macro*
The body of a **login-forms** is composed of forms to be evaluated, whose effects are to be undone if you log out. For example,

```
(login-forms
  (setq fs:*defaults-are-per-host* t))
```

would set the variable immediately but arrange for its previous value to be restored if you log out.

login-forms is not an AI program; it must be told how to undo each function that will be used immediately inside it. This is done by giving the function name (such as **setq**) a **:undo-function** property which is a function that takes a form as an argument and returns a form to undo the original form. For **setq**, this is done as follows:

```
(defun (setq :undo-function) (form &aux results)
  (do ((l (cdr form) (cddr l)))
      ((null l))
    (cond ((boundp (car l))
           (push '(setq ,(car l) ',(symeval (car l))) results))
          (t (push '(makunbound ',(car l)) results))))
  '(progn . ,results))
```

Undo functions are standardly provided for the functions **setq**, **pkg-goto-globally**, **setq-globally**, **add-initialization**, **deff**, **defun**, **defsubst**, **macro**, **advise** and **zwei:set-comtab**. Constructs which macroexpand into uses of those functions are also supported.

Note that setting ***read-base*** and ***print-base*** should be done with **setq-globally** rather than **setq**, since those variables are likely to be bound by the **load** function while the init file is executed.

login-setq {*variable value*}... *Macro*
login-setq is like **setq** except that it puts a **setq** form on **logout-list** to set the variables to their previous values. **login-setq** is obsolete; use **login-forms** around a **setq** instead.

login-eval *x*
login-eval is used for functions that are "meant to be called" from INIT files, such as **zwei:set-comtab-return-undo**, which conveniently return a form to undo what they did. **login-eval** pushes the result of the form *x* onto **logout-list**. It is obsolete now because **login-forms** is a cleaner interface.

si:undoable-forms-1 *undo-list-name forms* &optional *complaint-string*
This is what **login-forms** uses. *forms* is a list of forms; they are evaluated and forms for undoing their effects are pushed onto the value of the symbol *undo-list-name*. If an element of *forms* has no known way to be undone, a message is printed using the string *complaint-string*. For **login-forms**, the string supplied is **"at logout"**.

35.9 Dribble Files

dribble &optional *filename*
With an argument, **dribble** opens *filename* as a 'dribble file' (also known as a 'wallpaper file') and then enters a Lisp listen loop in which ***standard-input*** and ***standard-output*** are rebound to direct all the output and echoing they do to the file as well as to the terminal.

Dribble output can be sent to an editor buffer by using a suitable pathname; see section 24.7.6, page 575.

Calling **dribble** with no arguments terminates dribbling; it throws to the original call to **dribble**, which closes the file and returns.

dribble-all &optional *filename*

Like **dribble** except that all input and output goes to the dribble file, including break loops, queries, warnings and sessions in the debugger. This works by binding ***terminal-io*** instead of ***standard-output*** and ***standard-input***.

35.10 Version Information

Common Lisp defines several standard ways of inquiring about the identity and capabilities of the Lisp system you are using.

features *Variable*

A list of atoms which describe the software and hardware features of the Lisp implementation. By default, this is

```
(:loop :defstruct :lispm :cadr :mit :chaos :sort :fasload :string
 :newio :roman :trace :grindef :grind :common)
```

Most important is the symbol **:lispm**; this indicates that the program is executing on the Lisp Machine. **:cadr** indicates the type of hardware, **:mit** which version of the Lisp Machine operating system, and **:chaos** that the Chaosnet protocol is available. **:common** indicates that Common Lisp is supported.

Most of the other elements are for Maclisp compatibility. Common Lisp defines the variable ***features*** but does not define what should appear in the list. The order of elements in the list has no significance. Membership checks should use **string-equal** so that packages are not significant

The #+ and #- read constructs (page 525) check for the presence of an element in this list. Thus, **#+lispm** when read by a Lisp Machine causes the following expression to be significant, because **:lispm** is present in the features list.

The remaining standard means of inquiry are specified by Common Lisp to be functions rather than variables, for reasons that seem poorly thought out.

lisp-implementation-type

Returns a string saying what kind of Lisp implementation you are using. On the Lisp Machine it is always **"Zetalisp"**.

lisp-implementation-version

Returns a string saying the version numbers of the Lisp implementation. On the Lisp Machine it looks something like

"System 98.3, CADR 3.0, ZMAIL 52.2".

machine-type

Returns a string describing the kind of hardware in use. It is **"CADR"** or **"LAMBDA"**.

machine-version
Returns a string describing the kind of hardware and microcode version. It starts with the value of machine-type. It might be "CADR Microcode 309".

machine-instance
Returns a string giving the name of this machine. Do not be confused; the value is a string, not an instance. Example: "CADR-18".

software-type
Returns a string describing the type of operating system software that Lisp is working with. On the Lisp Machine, it is always "Zetalisp", since the Lisp Machine Lisp software is the operating system.

software-version
Returns a string describing the version numbers of the operating system software in use. This is the same as lisp-implementation-version on the Lisp Machine since the same software is being described.

short-site-name
Returns a string giving briefly the name of the site you are at. A site is an institution which has a group of Lisp Machines. The string you get is the value of the :short-site-name site option as given in SYS: SITE; SITE LISP. See section 35.12, page 810 for more information. Example: "MIT AI Lab".

long-site-name
Returns a string giving a verbose name for the site you are at. This string is specified by the site option :long-site-name. Example: "Massachusetts Institute of Technology, Artificial Intelligence Laboratory".

35.11 Booting and Disk Partitions

A Lisp Machine disk is divided into several named *partitions* (also called *bands* sometimes). Partitions can be used for many things. Every disk has a partition named PAGE, which is used to implement the virtual memory of the Lisp Machine. When you run Lisp, this is where the Lisp world actually resides. There are also partitions that hold saved images of the Lisp Machine microcode, conventionally named MCR*n* (where *n* is a digit), and partitions that hold saved images of Lisp worlds, conventionally named LOD*n*. A saved image of a Lisp world is also called a *virtual memory load* or *system load*. The microcode and system load are stored separately so that the microcode can be changed without going through the time-consuming process of generating a new system load.

The directory of partitions is in a special block on the disk called the label. The label names one of the partitions as the current microcode and one as the current system load. When you cold-boot, the contents of the current microcode band are loaded into the microcode memory, and then the contents of the current saved image of the Lisp world is copied into the PAGE partition. Then Lisp starts running. When you warm-boot, the contents of the current microcode band are loaded, but Lisp starts running using the data already in the PAGE partition.

For each partition, the directory of partitions contains a brief textual description of the contents of the partition. For microcode partitions, a typical description might be "UCADR 310"; this means that version 310 of the microcode is in the partition. For saved Lisp images, it is a little more complicated. Ideally, the description would say which versions of which systems are loaded into the band. Unfortunately, there isn't enough room for that in most cases. A typical description is "99.4 Daed 5.1", meaning that this band contains version 99.4 of System and version 5.1 of Daedalus. The description is created when a Lisp world is saved away by disk-save (see below).

35.11.1 Manipulating the Label

print-disk-label &optional (*unit* 0) (*stream* *standard-output*)

Prints a description of the label of the disk specified by *unit* onto *stream*. The description starts with the name of the disk pack, various information about the disk that is generally uninteresting, and the names of the two current load partitions (microcode and saved Lisp image). This is followed by one line of description for each partition. Each one has a name, disk address, size, and textual comment. The current microcode partition and the current system load partition are marked with asterisks, each at the beginning of the line.

unit may be the unit number of the disk (most Lisp machines just have one unit, number 0), or the host name of another Lisp Machine on the Chaosnet, as a string (in which case the label of unit 0 on that machine is printed, and the user of that machine is notified that you are looking at his label), or, for CADRs only, the string "CC" (which prints the label of unit 0 of the machine connected to this machine's debugging hardware).

Use of "CC" as the *unit* is the way to examine or fix up the label of a machine which cannot work because of problems with the label. On a Lambda, this must be done through the SDU.

set-current-band *partition-name* &optional (*unit* 0)

Sets the current saved Lisp image partition to be *partition-name*. If *partition-name* is a number, the name LOD*n* is used.

unit can be a disk drive number, the host name of another Lisp Machine, or the string "CC". See the comments under print-disk-label, above.

If the partition you specify goes with a version of microcode different from the one that is current, this function offers to select the an appropriate microcode partition as well. Normally you should answer Y.

set-current-microload *partition-name* &optional (*unit* 0)

Sets the current microcode partition to be *partition-name*. If *partition-name* is a number, the name MCR*n* is used.

unit can be a disk drive number, the host name of another Lisp Machine, or the string "CC". See the comments under print-disk-label, above.

si:current-band &optional (*unit* 0)
si:current-microload &optional (*unit* 0)

Return, respectively, the name of the current band and the current microload on the specified unit.

When using the functions to set the current load partitions, be extra sure that you are specifying the correct partition. Having done it, cold-booting the machine will reload from those partitions. Some versions of the microcode will not work with some versions of the Lisp system, and if you set the two current partitions incompatibly, cold-booting the machine will fail. To fix this, on a CADR, use another CADR's debugging hardware, running print-disk-label and set-current-band on the other CADR and giving "CC" as the *unit* argument. On a Lambda, this is done via the SDU.

si:edit-disk-label *unit* &optional *init-p*

Runs an interactive label editor on the specified unit. This editor allows you to change any field in the label. The Help key documents the commands. You have to be an expert to need this and to understand what it does, so the commands are not documented here. Ask someone if you need help. You can screw yourself very badly with this function.

disk-restore &optional *partition*

Allows booting from a band other than the current one. *partition* may be the name or the number of a disk partition containing a virtual-memory load, or nil or omitted, meaning to use the current partition. The specified partition is copied into the paging area of the disk and then started.

Although you can use this to boot a different Lisp image than the installed one, this does not provide a way to boot a different microcode image. disk-restore brings up the new band with the currently running microcode.

disk-restore asks the user for confirmation before doing it.

describe-partition *partition* &optional *unit*

Tells you various useful things about a partition; including where on disk the partition begins, and how long it is.

If you specify a saved Lisp system partition, such as LOD3, it also tells you important information about the contents of the partition: the microcode version which the partition goes with, the size of the data in the partition and the highest virtual address used. The size of the partition tells how large a partition you need to make a copy of this one, and the highest virtual address used (which is measured in units of disk blocks) tells you how large a PAGE partition you need in order to run this partition.

35.11.2 Updating Software

Of all the procedures described in this section, the most common one is to take a partition containing a Lisp image, update it to have all the latest patches (see section 28.8, page 672), and save it away in a disk partition. The function **load-and-save-patches** does it all conveniently for you.

load-and-save-patches

Loads patches and saves a band, with a simple user interface. Run this function immediately after cold booting, without logging in first; it logs in automatically as **LISPM** (or whatever is specified in the site files). The first thing it does is print the list of disk partitions and ask you which one to save in. Answer **LOD***n*, using the name of a partition from the printed list. You must then confirm. Then the patches are loaded and the resulting world is saved with no further user interaction, as long as no problem arises.

It is convenient to use this function just before you depart, allowing it to finish unattended.

If you wish to do something other than loading all and only the latest patches, you must perform the steps by hand. Start by cold-booting the machine, to get a fresh, empty system. Next, you must log in as something whose INIT file does not affect the Lisp world noticably (so that when you save away the Lisp image, the side-effects of the INIT file won't get saved too); on MIT-OZ, for example, you can log in as **LISPM** with password **LISPM**. Now you can load in any new software you want; usually you should also do **(load-patches)** for good measure. You may also want to call **si:set-system-status** to change the release status of the system.

When you're done loading everything, do **(print-disk-label)** to find a band in which to save your new Lisp world. It is best not to reuse the current band, since if something goes wrong during the saving of the partition, while you have written, say, half of the band that is current, it may be impossible to cold-boot the machine. Once you have found the partition, you use the **disk-save** function to save everything into that partition.

disk-save *partition-name* &optional *no-query incremental*

Saves the current Lisp world in the designated partition. *partition-name* may be a partition name (a string), or it may be a number in which case the name **LOD***n* is used.

The user is first asked for yes-or-no confirmation that he really wants to reuse the named partition. A non-**nil** value for *no-query* prevents this question. This is only for callers that have already asked.

Next it is necessary to figure out what to put into the textual description of the band, for the disk label. This starts with the brief version of **si:system-version-info** (see page 674). Then comes a string of additional information; if *no-query* is **nil**, the user is offered the chance to provide a new string. The current value of this string is returned by **si:system-version-info** and printed by booting. The version info and the string both go in the comment field of the disk label for this band. If they don't together fit into the fixed size available, the user is asked to retype the whole thing (the version info as well as your comment) in a compressed form that does fit.

The Lisp environment is then saved away into the designated partition, and then the equivalent of a cold-boot from that partition is done.

Once the patched system has been successfully saved and the system comes back up, you can make it current with **set-current-band**.

When you do a **disk-save**, it may tell you that the band you wish to save in is not big enough to hold all the data in your current world. It may be possible for you to reduce the size of the data so that it will fit in that band, by garbage collecting. Simply do **(si:full-gc)**.

Try to avoid saving patched systems after running the editor or the compiler. This works, but it makes the saved system a lot bigger. In order to produce a clean saved environment, you should try to do as little as possible between the time you cold-boot and the time you save the partition.

si:login-history *Variable*

The value of **si:login-history** is a list of entries, one for each person who has logged into this world since it was created. This makes it possible to tell who **disk-saved** a band with something broken in it. Each entry is a list of the user ID, the host logged into, the Lisp Machine on which the world was being executed, and the date and time.

35.11.3 Saving Personal Software

If you have a large application system which takes a while to load, you may wish to save a band containing it.

To do this, boot a fresh band, log in without running your init file, do **make-system** to load the application system, and then invoke **disk-save**. When **disk-save** asks for an additional comment, give your name or the name of the application system you loaded, and a date. This will tell other people who to ask whether the band is still in use if they would like to save other things.

You can greatly reduce the amount of disk space needed for the saved band by making it an *incremental* band; that is, a band which contains the differences between the Lisp world you want to save and the system band you originally loaded. Since all the pages of the system which your application program did not change do not have to be saved, an incremental band is generally much smaller—perhaps by a factor of ten.

To make an incremental band, give a non-nil third argument to **disk-save**, as in

```
    (disk-save "lod4" nil t)
```

Figuring out which pages need to be saved in the incremental band takes a couple of extra minutes.

You can restore the incremental band with **disk-restore** or boot it like any other band. This works by first booting the original band and then copying in the differences that the incremental band records. It takes only a little longer than booting the original system band.

The original band to which an incremental band refers must be a complete load. When you update a standard system band (loading patches, for instance) you should always make a complete load, so that the previous system band is not needed for the new one to function.

The incremental band records the partition name of the original system band. That original band must still exist, with the same contents,. in order for the incremental band to work properly. The incremental band contains some error check data which is used to verify this. The error checking is done by the microcode when the incremental band is booted, but it is also done by set-current-band, so that you are not permitted to select an incremental band if it is not going to work.

When using incremental bands, it is important to preserve the system bands that they depend on. Therefore, system bands should not be updated too frequently. describe-partition on an incremental band says which full band it depends on; you can use this to determine which bands should be kept for the sake of incremental bands that depend on them.

In order to realize the maximum savings in disk space possible because of incremental bands, you must make the partition you saved in smaller once the save is finished and you know how much space was actually used. This is done with si:edit-disk-label. The excess space at the end of the partition can be used to make another partition which is used for the next incremental band saved. Eventually when some of the incremental bands are no longer needed the rest must be shuffled so that the free space can be put together into larger partitions. This can be done with si:copy-disk-partition.

An easier technique is to divide a couple of the initial partitions into several equal-sized partitions of about 4000 pages, and use these for all incremental saving. You can easily provide room for 12 incremental bands this way in addition to a few system bands and file system.

You must not do a garbage collection to reduce the size of the world before you make an an incremental band. This is because garbage collection alters so many pages that an incremental band would be as big as a complete band.

35.11.4 Copying Bands

The normal way to install new software on a machine is to copy the microcode and world load bands from another machine.

The first step is to find a machine that is not in use and has the desired system. Let us call this the source machine. The machine where the new system is to be installed is the target machine. You can use finger to see which machines are free, and use print-disk-label with an argument to examine the label of that machine's disk and see if it has the system you want.

Then you should do a (print-disk-label) to find suitable partitions to copy them into. It is advisable not to copy them into the selected partitions; if you did that, and the machine crashed in the middle, you would be unable to boot it.

Before copying a band from another machine, double-check the partition names by printing the labels of both machines, and make sure no one is using the other machine. Also double-check with describe-partition that the world load and microcode go together. Then use this function:

si:receive-band *source-host source-band target-band* &optional *subset-start subset-size*

Copies the partition on *source-host*'s partition named *source-band* onto the local machine's partition named *target-band*. This takes about ten minutes. It types out the size of the partition in pages, and types a number every 100 pages telling how far it has gotten. It displays an entry in the who line on the remote machine saying what's going on.

The *subset-start* and *subset-size* arguments can be used to transfer only part of a partition. They are measured in blocks. The default for the first is zero, and the default for the second is to continue to the end of the data in the band. These arguments are useful for restarting a transfer that was aborted due to network problems or a crash, based on the count of hundreds of blocks that was printed out before the crash.

To go the other direction, use **si:transmit-band**.

si:transmit-band *source-band target-host target-band* &optional *subset-start subset-size*

This is just like **si:receive-band**, except you use it on the source machine instead of the target machine. It copies the local machine's partition named *source-band* onto *target-machine*'s partition named *target-band*.

It is preferable to use **si:receive-band** so that you are present at the machine being written on.

After transferring the band, it is good practice to make sure that it really was copied successfully by comparing the original and the copy. All of the known reasons for errors during band transfer have (of course) been corrected, but peace of mind is valuable. If the copy was not perfectly faithful, you might not find out about it until a long time later, when you use whatever part of the system that had not been copied properly.

si:compare-band *source-host source-band target-band* &optional *subset-start subset-size*

This is like **si:receive-band**, except that it does not change anything. It compares the two bands and complains about any differences.

Having gotten the current microcode load and system load copied into partitions on your machine, you can make them current for booting using **set-current-band**.

35.12 Site Options and Host Table

The Lisp Machine system has options that are set at each site. These include the network addresses of other hosts, which hosts have file servers, which host to find the system source files and patch files on, where to send bug reports, what timezone the site is located in, and many other things.

The per-site information is defined by three files: SYS: SITE; SITE LISP, SYS: SITE; LMLOCS LISP, and SYS: CHAOS; HOSTS TXT.

SYS: CHAOS; HOSTS TXT is the network host table. It gives the names and addresses of all hosts that are to be known to the Lisp Machine for any purposes. It also says what type of machine the host is, and what operating system runs on it.

SYS: SITE; LMLOCS LISP specifies various information about the Lisp Machines at your site, including its name, where it is physically located, and what the default machine for logging in should be.

SYS: SITE; SITE LISP specifies all other site-specific information. Primarily, this is contained in a call to the special form **defsite**.

defsite *site-name* (*site-option value*)... *Macro*

This special form defines the values of site-specific options, and also gives the name of the site. Each *site-option* is a symbol, normally in the keyword package, which is the name of some site option. *value* is the value for that option; it is evaluated. Here is a list of standardly defined site options:

:sys-host
: The value is a string, the name of the host on which the system source files are stored. This host becomes the translation of logical host SYS.

:sys-host-translation-alist
: The value is an alist mapping host names into translation-list variables. Each translation list variable's value should be an alist suitable for being the third argument to fs:add-logical-pathname-host (see page 574). The car of an element may be nil instead of a host name; then this element applies to all hosts not mentioned.

The normal place to find the system sources is on the host specified by the :sys-host keyword, in the directories specified by the translation list variable found by looking that host up in the value of the :sys-host-translation-alist keyword. If you specify a different host as the system host with si:set-sys-host, that host is also looked up in this alist to find out what directories to use there.

Here is what is used at MIT:

```
(defsite :mit
  ...
  (:sys-host-translation-alist
    '(("AI" . its-sys-pathname-translations)
      ("OZ" . oz-sys-pathname-translations)
      ("FS" . its-sys-pathname-translations)
      ("LM" . its-sys-pathname-translations)
      (nil . its-sys-pathname-translations)))
  ...)
```

```
(defconst oz-sys-pathname-translations
  '(("CC;" "<L.CC>")
    ("CHAOS;" "<L.CHAOS>")
    ("DEMO;" "<L.DEMO>")
    ...
    ("SITE;" "<L.SITE>")
    ("SYS;" "<L.SYS>")
    ("SYS2;" "<L.SYS2>")
    ...
    ("ZMAIL;" "<L.ZMAIL>")
    ("ZWEI;" "<L.ZWEI>")
    ))
```

:sys-login-name
:sys-login-password
: These specify the username and password to use to log in automatically to read system patch files, microcode symbol tables and error tables. The values should be strings.

:chaos
: nil if the site has no Chaosnet; otherwise, a string, the name of the Chaosnet that the site is on. Names for Chaosnets will eventually be used to permit communication between Chaosnets, probably through special gateway servers. Except when multiple sites are on a single Chaosnet, normally the Chaosnet name should be the same as the site name (but as a string, not a symbol).

:standalone
: The value should be t for a Lisp Machine that is operated without a network connection. This causes the Lisp Machine to not to try to use the Chaosnet for getting the time. On the Lambda, the time will obtained from the SDU's clock. On the CADR, the time will be obtained from the user.

:default-associated-machine
: This should be a string which is the name of a host to use as the associated host for any Lisp Machine not mentioned in the LMLOCS file.

:usual-lm-name-prefix
: This should be a string which is the typical beginning of host names of Lisp Machines at your site. At MIT, it is "CADR-".

:chaos-file-server-hosts
: This should be a list of names of hosts that have file servers, including Lisp Machines which other Lisp Machines should know about.

:lmfile-server-hosts
: This should be a list of names of Lisp Machines that provide servers for the LMFILE file system. The entry for such a machine should be one of the nicknames of that machine. By virtue of its presence in this list, it becomes the name by which the LMFILE file system there can be accessed remotely.

:chaos-time-server-hosts
: This should be a list of names of hosts that support TIME servers. These

are hosts that the Lisp Machine can ask the time of day from when you boot.

:chaos-host-table-server-hosts
: This should be a list of names of hosts that support host-table servers, which can be used to inquire about hosts on networks that the Lisp Machine does not know about in its own host table.

:chaos-mail-server-hosts
: This should be a list of names of hosts that support mail servers which are capable of forwarding mail to any known host.

:timezone
: This should be a number, the number of hours earlier than GMT of standard time in the timezone where this site is located.

:host-for-bug-reports
: This should be a string, the name of the host at which bug-report mailboxes are located.

:local-mail-hosts
: This should be a list of names of hosts that ZMail should consider "local" and omit from its summary display.

:spell-server-hosts
: This should be a list of hosts that have spelling corrector servers.

:comsat
: This should be t if mail can be sent through the COMSAT mail demon. This is true only at MIT.

:default-mail-mode
: This should be the default mode for use in sending mail. The options are :file (use COMSAT), :chaos (use one of the :chaos-mail-server-hosts), or :chaos-direct (like :chaos, but go direct to the host that the mail is addressed to whenever possible).

:gmsgs
: This should be t if GMSGS servers are available.

:arpa-gateways
: This should be a list of names of hosts that can be used as gateways to the Arpanet. These hosts must provide a suitable Chaosnet server which will make Arpanet connections. It should be nil if your site does not have an Arpanet connection.

:arpa-contact-name
: If you have Arpanet gateways, this is the Chaosnet contact name to use. Nowadays, it should be "TCP".

:dover
: This should be t if your site has a Dover printer.

:default-printer
: This should be a keyword which describes the default printer for hardcopy commands and functions to use. Possible values include :dover, nil, or any other printer type that you define (see section 35.2, page 785).

:default-bit-array-printer
: Like :default-printer, but this is the default for only hardcopy-bit-array

to use.

:esc-f-arg-alist
: This says what various numeric arguments to the **Terminal F** command mean. It is a list of elements, one for each possible argument. The car of an element is either a number or **nil** (which applies to **Terminal F** with no argument). The cdr is either **:login** (finger the login host), **:lisp-machines** (finger all Lisp Machines at this site), **:read** (read some hosts from the keyboard), or a list of host names.

:verify-lm-dumps
: If the value is **t**, Lisp Machine file system dump tapes are verified.

Other site options are allowed, and your own software can look for them.

35.12.1 Updating Site Information

To update the site files, you must first recompile the sources. Do this by

```
    (make-system 'site 'compile)
```

This also loads the site files.

To just load the site files, assuming they are compiled, do

```
    (make-system 'site)
```

load-patches does that automatically.

You should never load any site file directly. All the files must be loaded in the proper fashion and sequence, or the machine may stop working.

35.12.2 Accessing Site Options

Programs examine the site options using these variables and functions:

site-name *Variable*

The value of this variable is the name of the site you are running at, as defined in the defsite in the SITE file. You can use this in run-time conditionals for various sites.

get-site-option *keyword*

Returns the value of the site option *keyword*. The value is **nil** if *keyword* is not mentioned in the SITE file.

define-site-variable *variable keyword* [*documentation*] *Macro*

Defines a variable named *variable* whose value is always the same as that of the site option *keyword*. When new site files are loaded, the variable's value is updated. *documentation* is the variable's documentation string, as in **defvar**.

define-site-host-list *variable keyword [documentation]* *Macro*

Defines a variable named *variable* whose value is a list of host objects specified by the site option *keyword*. The value actually specified in the SITE file should be a list of host names. When new site files are loaded, the variable's value is updated. *documentation* is the variable's documentation string, as in **defvar**.

35.12.3 The LMLOCS File

The LMLOCS file contains an entry for each Lisp Machine at your site, and tells the system whatever it needs to know about the particular machine it is running on. It contains one form, a defconst for the variable machine-location-alist. The value should have an element for each Lisp Machine, of this form:

```
("MIT-LISPM-1"  "Lisp Machine One"
 "907 [Son of CONS] CADR1's Room x6765"
 (MIT-NE43 9) "OZ" ((:default-printer :dover)))
```

The general pattern is

(*host-full-name pretty-name*
 location-string
 (*building floor*) *associated-machine site-options*)

The *host-full-name* is the same as in the host table.

The *pretty-name* is simply for printing out for users on certain occasions.

The *location-string* should say where to find the machine's console, preferably with a telephone number. This is for the FINGER server to provide to other hosts.

The *building* and *floor* are a somewhat machine-understandable version of the location.

The *associated-machine* is the default file server host name for login on this Lisp Machine.

site-options is a list of site options, just like what goes in the **defsite**. These site options apply only to the particular machine, overriding what is present in the SITE file. In our example, the site option :default-printer is specified as being :dover, on this machine only.

si:associated-machine *Variable*

The host object for the associated machine of this Lisp Machine.

Concept Index

Flavor Index

Operation Index

Keyword Index

Object Creation Options

Meter Index

Variable Index

Function Index

Condition Name Index