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

        #�in*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 #ɔ 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.