INTRODUCTION TO FLAVORS

Table of Contents

1. Introduction

Object oriented programming is available in Common Lisp. 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 Common Lisp, and when you should use them. It assumes no prior knowledge of any other languages.

2. 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 with which "piece of text" the window is associated.

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 the operations that can be performed on any of the instances of each type.

This should not be wholly unfamiliar to the reader. Earlier in this manual, we saw a few examples of this kind of programming. A simple example is disembodied property lists, and the functions get, putprop, and remprop. The disembodied property list is a type of object; you can instantiate one with (cons nil nil) (that is, by evaluating this form you can create a new disembodied property list); there are three operations on the object, namely get, putprop, and remprop. Another example in the manual was the first example of the use of defstruct, which was called a ship. defstruct automatically defined some operations on this object: the operations to access its elements. We could define other functions that did useful things with ships, 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 property list case, the Lisp object is a list with alternating indicators and values; 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-mass 5.0)) alters it.

We have now seen the essence of object-oriented programming. A conceptual object is modelled 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.

3. 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, it helps and encourages 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 putprops 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)
  ship-x-position
  ship-y-position
  ship-x-velocity
  ship-y-velocity
  ship-mass)

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

(defun ship-direction (ship)
  (atan (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)
  ship-x-position
  ship-y-position
  ship-speed
  ship-direction
  ship-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 works, because 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 identity (eqness)) 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 which 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 is what specifies the interface between the two modules.

Unlike some other languages that provide abstract types, Common Lisp makes no attempt to have the language automatically forbid constructs that circumvent

the contract. This is intentional. One reason for this is that Lisp 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 in Lisp; users are allowed to get into any part of the language system and change what they want to change.

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).

It is just as true of such abstract-operation functions as of ordinary functions that sometimes they are simple enough that we want the compiler 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, defsubsts, 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 "outside-accessible instance variables" feature is used; see the section on the outside-accessible-instance-variables option, where this problem is explained further. If you don't use the "outside-accessible instance variables" feature, you don't have to worry about this.

4. 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 ships 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 will need to know the ship's attributes, and will have to 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 Common Lisp. The + function will accept fixnums, flonums, ratios, bignums, etc., and perform the appropriate type 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 ships, meteors, 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.

A terminology for the use of such generic operations has emerged from the Smalltalk and Actor languages: performing a generic operation is called sending a message. The objects in the program are thought of as little people, who get sent messages and respond with answers. In the example above, the objects are sent x-position messages, to which they respond with their x position. This message passing is how generic operations are performed.

Sending a message is a way of invoking a function. Along with the name of the message, in general, some arguments are passed; when the object is done with the message, some values are returned. The sender of the message is simply calling a function with some arguments, and getting some values back. The interesting thing is that the caller did not specify the name of a procedure to call. Instead, it specified a message name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke was found from this information.

When a message is sent to an object, a function therefore must be found to handle the message. The two data used to figure out which function to call are the type of the object, and the name of the message. 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 name are data which are passed as arguments to the function, so the name 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 kind of message to 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 named x-position (with 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.

5. Generic Operations in Lisp

How do we implement message passing in Lisp? By convention, objects that receive messages are always functional objects (that is, you can apply them to arguments), and a message is sent to an object by calling that object as a function, passing the name of the message as the first argument, and the arguments of the message as the rest of the arguments. Message names are represented by symbols; normally these symbols are in the keyword package 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:

```
(funcall my-ship ':x-position)
```

This form returns the x position as its returned value. To set the ship's x position to 3.0, we send it a message like this:

```
(funcall 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 a message name. The object must consider this message name, find the function which is the method for that message name, and invoke that function.

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; objects can't just be defstructs any more, since those aren't functions. But the structure defined by defstruct was doing something useful: it was holding the instance variables (the internal state) of the object. We need a function with internal state; that is, we need a coroutine.

Of the features presented so far, the most appropriate is the closure. 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.

While using closures 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 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 to do are very easy to do with flavors. The rest of this chapter describes flavors.

6. 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) ()
  (atan 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 message is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the messages :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 messages named :speed. The second subform is the lambda-list, and the rest is the body of the function that handles the :speed message. The body can refer to or set any instance variables of the flavor, the same as it can with local variables or special variables. When any instance of the ship flavor is invoked with a first argument of :direction, the body of the second defmethod will be 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 when the arguments of atan are evaluated, the values of instance variables of the object to which the message was sent will be used as the arguments. atan will be invoked, and the result it returns will be returned by the instance itself.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor will have 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 will return an object whose printed representation is:

```
#<SHIP 1213731>
```

(Of course, the value of the magic number will vary; it is not interesting anyway.) The argument to make-instance is, as you can see, 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 messages named :set-x-position, :set-y-position, and so on; each such method takes one argument, and sets the corresponding instance variable to the given value.

Another option we can add to the defflavor is :initable-instance-variables, to allow us to initialize the values of the instance variables when an instance is first created. :initable-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 set of init options are sometimes called the init-plist because they are 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
  :initable-instance-variables)
```

All we have to do is evaluate this new defflavor, and the existing flavor definition will be updated and now include the new methods and initialization options. In fact, the instance we generated a while ago will now be able to accept these new messages! We can set the mass of the ship we created by evaluating

```
(funcall my-ship ':set-mass 3.0)
```

and the mass instance variable of my-ship will properly get set to 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:         unbound
        Y-POSITION:         unbound
        X-VELOCITY:         unbound
        Y-VELOCITY:         unbound
        MASS:               3.0
```

Now that the instance variables are "initable", 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:         unbound
        Y-VELOCITY:         unbound
        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
  :initable-instance-variables)

(setq another-ship (make-instance 'ship ':x-position 3.4))

(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:               unbound
```

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 unbound.

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.

7. 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 defmethods 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)
   :initable-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 will still accept the :x-velocity and :speed messages, and they will do the same thing. However, the :engine-power message will also be understood (and will return 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, and eliminating duplicates. 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 will 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 will have an instance variable named foo, and any methods that refer to foo will 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, and 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 message supported by the flavor. This function is constructed out of all the methods for that message 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 a message by providing your own method. Your method will be called, and foo's will never be called.

Simple overriding is often useful; if you want to make a new flavor bar that is just like foo except that it reacts completely differently to a few messages, then this will work. 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 message, 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.

When methods are combined, a single primary method is found; it comes from the first component flavor that has one. Any primary methods belonging to later component flavors are ignored. This is just what we saw above; bar could override foo's primary method by providing its own primary method.

However, you can define other kinds of methods. In particular, you can define daemon methods. They 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 message, 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 will run before any of those in foo, and bar's after-daemons will 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) prints instances of flavors by sending them :print-self messages. The first argument to the :print-self message 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 message, then that method will take over the job of printing completely; vanilla-flavor's method will not be called at all. However, if we give ship a before-daemon method for the :print-self message, then it will get invoked before the vanilla-flavor message, 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 on method-combination. The vanilla-flavor and what it does for

you are also explained later in a section on the vanilla flavor.

8. The Instance Datatype

Various parts of what are considered Flavors require hooks into the system. Portable code should attempt not to use any of the features mentioned in this section, with the exception of :print-self, which can be portably implemented using defstruct.

One traditional hook is into the type system. (type-of instance) returns the flavor that the instance was made from. typep instance flavor) returns T iff the instance's flavor has the flavor as a component. Lastly, (instancep instance) <=> (typep instance 'instance) returns T.

Various lisp operations on an instance should simply send a message to the object. It has been proposed that the system should only send the names of the functions, rather than any keyworded symbol, but vanilla-flavor could translate. Describe of an instance may send a :describe message to the instance. More esoteric hooks would be having the system send a :fasd-form or some such message to get a form that, when loaded and evaluated, would make a copy of the object; and having eval of an instance send an EVAL message to the object.

9. Flavor Functions

all-flavor-names [Variable]

This is a list of the names of all the flavors that have ever been defflavored.

undefined-flavor-names [Variable]

This is a list of all flavors which have been referred to but not defined.

flavor-compile-methods [Variable]

If this variable is non-nil, combined methods are automatically compiled.

dirty-flavors [Variable]

A stack (implemented as a vector) of unclean flavor names; that is, those which need to be updated.

defflavor name ({var}*) ({flavor}*) {option}* [Macro]

Flavor-name is a symbol which serves to name this flavor. Traditionally the flavor definition is a defstruct on the flavor property of the name.

(typep obj), where obj is an instance of the flavor named flavor-name, will return 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 will be 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 will remain unbound.

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 the section on defflavor-options.

defmethod (name method-type message) lambda-list {form}* [Macro]
Defines a method, that is, a function to handle a particular message sent to an instance of a particular flavor. Flavor-name is a symbol which is the name of the flavor which is to receive the method. Method-type is a keyword symbol for the type of method; it is omitted when you are defining a primary method, which is the usual case. Message is a keyword symbol which names the message to be handled.

The meaning of the method-type depends on what kind of method-combination is declared for this message. For instance, for daemons :before and :after are allowed. See the section on method combination 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 message keyword, is automatically handled, and so it is not included in the lambda-list. form1, form2, etc. are the function body; the value of the last form is returned.

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

make-instance name {init-option value} [Function]
This 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. If the flavor supports the :init message, it 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.

send object message-name &rest arguments [Function]
Finds the appropriate handler for the message and invokes it with the given arguments, and some additiona implementation-dependent arguments.

defwrapper (flavor message) (({argname}*) . bodyname) . body [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 which expands into code which 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 message to the bar flavor, which takes two arguments, and you have a lock-frobboz special-form which knows how to lock the lock (presumably it generates an unwind-protect). lock-frobboz needs to see the first argument to the message; 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 defwrappered macro from knowing the exact implementation and allows several defwrappers 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))          ;Do nothing if arg1 is nil
         (t before-code
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular message; 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 message and used free by the methods.

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

Similarly you might want a wrapper which puts a *catch around the processing of a message 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 will execute within that wrapper's context.

defwhopper (flavor [type] message) arglist . body [Special Form]
Arglist is the same as the argument list for any method handling message. The default type is :whopper.

Whoppers are to functions as wrappers are to macros. Code for wrappers might be duplicated many times, and so whoppers were devised to save space. Note that to do this, the whopper code is a function, and so must not only be called, but it must call a continuation function, for a net cost of two function calls.

Whoppers have three forms that they may use in order to continue the combined method:

continue-whopper &rest args [Macro]
Calls the methods for the message that the whopper is handling. Args is the list of arguments sent. This only works inside a whopper. The whopper may change the arguments rather than passing those it receives verbatim.

lexpr-continue-whopper &rest args [Macro]
Uses apply instead of funcall to call the continuation function.

continue-whopper-all [Function]
This performs a whopper continuation and simply passes the arguments it gets all on the the methods. This avoids consing a rest argument.

Whoppers may be considered a kind of wrapper, for the purposes of ordering. If a flavor defines both a wrapper and a whopper, though, the wrapper goes outside the whopper.

undefmethod (flavor [type] message) [Macro] Generic undefining form. To undefine a wrapper, use it with :wrapper as the method type. For whoppers, use type :whopper.

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). Note that this is a lexical variable, not a special.

recompile-flavor name &optional message (do-dependents t)
Used to recalculate the combined methods for a flavor. Generally this is done for you automatically, but if a user macro or other such information unknown to the system changes, you may want to recalculate the combined methods explicitly. It's also possible that the system may miss a wrapper change if

it just hashes the body, in which case you'll have to do this manually. recompile-flavor only affects flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins.

compile-flavor flavor [Function]
Prepares the named flavor for instantiation; mainly, this means it calculates the combined methods for the flavor.

compiler-compile-flavors &rest flavors [Macro]
When placed in a file that defines some instantiable (or abstract) flavors, includes the code for the combined methods. It also makes sure that everything referred to by the combined method is present in the loadtime environment (assuming the same flavor structure). Also, when the sfasl file is loaded, all the other structures required for instantiation will get generated. flavor) will get generated.

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 to use, since the need to invoke the compiler at run-time makes programs that use flavors slow the first time they are run. (The compiler will still be called 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 compiler-compile-flavors for flavors that are going to be instantiated. For a flavor that will never 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.

The compiler-compile-flavors 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.

get-handler-for object message [Function] Given an object and a message, will return that object's method for that message, 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. This is only an informational device.

flavor-allowed-init-keywords flavor [Function]
Returns a list of all symbols that are valid init options for flavor, sorted alphabetically.

flavor-allows-init-keyword-p name keyword [Function]
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 which contributes the support of that keyword.

symeval-in-instance instance symbol &optional no-error-p [Function]
This function is used to find the value of an instance variable inside a particular instance. Instance is the instance to be examined, and symbol is the instance variable whose value should be returned. If there is no such instance variable, an error is signalled, unless no-error-p is non-nil in which case nil is returned.

set-in-instance instance symbol value [Function]
This function is used to alter the value of an instance variable inside a particular instance. Instance is the instance to be altered, symbol is the instance variable whose value should be set, and value is the new value. If there is no such instance variable, an error is signalled.

10. 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 signalled. 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 causes a method :x to be generated for each instance variable x. :X gets the value of x.

:initable-instance-variables creates a :x init-option for make-instance for each instance variable x.

:Settable-instance-variables causes a method :set-x to be generated for each instance variable x. :Set-x sets the value of x to the supplied value. Using this option cause :gettable-instance-variables and :initable-instance-variables to take effect.

:Init-keywords should be a list of all keywords accepted by the flavor's :init handler, if it has one. This is used for error handling; before the :init message is sent the flavors system checks that all of the keywords in the init-plist are either members of the init-keywords list or initable instance variables.

:default-init-plist takes arguments which 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 entries that initialize instance variables are not added to the init-plist seen by the :init methods.

:required-instance-variables declares that any flavor incorporating this one which 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 which 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 will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-init-keywords takes as arguments a list of init keywords that must be supplied. It is an error to try to make an instance of this flavor or a flavor which depends on it unless you specifiy these keywords to make-instance or as a :default-init-plist option in some component flavor.

:required-methods takes as arguments names of messages which 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 messages. Typically this option appears in the defflavor for a base flavor. Usually this is used when a base flavor does a funcall-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, bui only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message be detected when the flavor is defined (which usually means at compile time) rather than at run time.

:required-flavors takes as arguments a list of flavors which any flavor built on this one must include (possibly indirectly) as components. This is different from listing the flavors as component flavors in that required flavors are not specified to appear in any particular place in the component flavors list. If you require a flavor you allow all instance variables which it declares to be accessed, and cause an error to be signalled when you instantiate a flavor which doesn't include the requred flavor.

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 (expt (/ (funcall-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))
```

which 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 signalled, 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 vanilla-flavor is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard messages which all objects are supposed to understand. These include :print-self, :describe, :which-operations, and several other messages. See the section on the vanilla-flavor.

If any component of a flavor specifies the :no-vanilla-flavor option, then

vanilla-flavor will not be included in that flavor. This option should not be used casually.

:ordered-instance-variables 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 which are specially known about by microcode, and 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.

:outside-accessible-instance-variables takes as arguments some instance variables which are to be accessible from "outside" of this object, that is from functions other than methods. A macro 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.

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 will have to be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error will be signalled if the argument to the accessor macro is not an instance or is an instance which 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 will compile a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only 3 or 4 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. Any functions that use these accessor macros will have to be recompiled if the number or order of instance variables in the flavor is changed. The system will not know automatically to do this recompilation. If you aren't very careful, you may forget to recompile something, and have a very hard-to-find bug. Because of this problem, and because using these macros is less elegant than sending messages, the use of this option is discouraged. In any case the use of these accessor macros should be confined to the module which owns the flavor, and the "general public" should send messages.

: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$) would cause it to be named get$v instead.

:method-combination declares the way that methods from different flavors will be combined. Each "argument" to this option is a list (type order message1 message2...). Message1, message2, etc. are names of messages whose methods are to be combined in the declared fashion. type is a keyword which is a defined type of combination; see the section on method-combination. Order is a keyword whose interpretation is up to type; 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 message. If no component specifies a type of method combination, then the default type is used, namely :daemon. If more than one component of a flavor specifies it, then they must agree on the specification, or else an error is signalled.

:documentation takes documentation for the flavor as arguments. The list is remembered on the flavor's property list as the :documentation property. The (loose) standard for what can be in this list is as follows; this may be extended in the future. A string is documentation on what the flavor is for; this may consist of a brief overview in the first line, then several paragraphs of detailed documentation. A symbol is one of the following keywords:

- A :mixin is a flavor that you may want to mix with others to provide a useful feature. An :essential-mixin is a flavor that must be mixed in to all flavors of its class, or inappropriate behavior will ensue. A :lowlevel-mixin is a mixin used only to build other mixins. A :combination is a combination of flavors for a specific purpose. A :special-purpose flavor is a flavor used for some internal or kludgey purpose by a particular program, which is not intended for general use.

11. Flavor Families

The following organization conventions are recommended for all programs that use flavors.

A base flavor is a flavor that defines a whole family of related flavors, all of which will have that base flavor as one of their components. Typically the base flavor includes things relevant to the whole family, such as instance variables, :required-methods and :required-instance-variables declarations, default methods for certain messages, :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 would 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 might still define its own flavors, in a simple way. The facility might provide a base flavor and a set of mixins, and the caller can combine these in various combinations depending on exactly what it wants, since the facility probably would 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 to select options of its component flavors and you can define one or two methods to customize it "just a little".

12. Vanilla Flavor

The messages 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 vanilla-flavor:

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 slashify-p [Message]

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 slashification is enabled (prin1 vs princ). 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 [Message]

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

:which-operations [Message]

The object should return a list of the messages it can handle. Vanilla-flavor generates the list once per flavor and remembers

it, minimizing consing and compute-time. If a new method is added, the list is regenerated the next time someone asks for it.

:operation-handled-p operation [Message]
operation is a message name. The object returns whether it has a handler for the specified message.

:get-handler-for operation [Message]
operation is a message name. The object should return the method it uses to handle operation. If it has no handler for that message, it should return nil. This is like the get-handler-for function, but, of course, you can only use it on objects known to accept messages.

:send-if-handles message &rest args [Message]
The object sends itself message with args if the object handles message, otherwise it just returns nil.

:unclaimed-message message &rest args [Message]
If there is no method for message but there is a handler for :unclaimed-message, the handler is invoked with arguments message and all of the args. This is just like :default-handler, which is an option to defflavor.

13. Method Combination

Methods can have a symbolic type, which is referred to by the particular method combination defined for that method name. The purpose of the method combination is to produce a combined method which will handle the given message. It does this by ordering the methods that it finds in some manner and wrapping invocations of them with various forms.

Basic Method Types

:primary is the type of method produced when no type is given to defmethod. Therefore, most combinations deal with it.

:default by convention takes the place of :primary methods if none exist. A base flavor can therefore define default handlers for various operations.

:wrapper is used internally to remember wrappers.

:whopper is used internally to remember whoppers.

:combined is used internally to remember combined methods (so they can be inherited).

Other method types are only used with certain forms of combination.

Predefined Combination Types

One specifies a method combination with the :method-combination defflavor option. One specifies the name of the combination and possibly some other information, which is usually either :base-flavor-first or :base-flavor-last.

:daemon :base-flavor-last is the default form of combination. All the :before methods are called, then a single primary method (whose values are returned by the send), then all the :after methods are

called. The ordering argument to the combination affects the order of the :before and :after daemons.

:progn
:and
:or
:list
:append
:nconc All the primary methods are called inside the appropriate form.

:daemon-with-or is like :daemon, only the primary method may be overridden by the :or method types. The order of the :or methods is controlled by the order argument to the :method-combination defflavor option. The combine dmethod looks something like this:

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

:daemon-with-and is like :daemon-with-and, only :and methods are wrapper in an and before the primary method.

:daemon-with-override is like :daemon-with-or, only the :override methods are wrapper in an or before everything else, including the before and after daemons.

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 highest level of abstraction is called, then :before daemons of successively lower levels of abstraction 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 lowest level of abstraction is called, followed by the :after daemons at successively higher levels of abstraction.

Now, if there is no interaction among all these methods, if their actions are completely orthogonal, then it doesn't matter whether you use a :before daemon or an :after daemon. It makes 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 message, 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 message 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.

Of course, since flavors are not hierarchically organized, the notion of levels of abstraction is not strictly applicable. However, it remains a useful way of thinking about systems.

Index