19. Defstruct

defstruct provides a facility in Lisp for creating and using aggregate datatypes with named elements. These are like "structures" in PL/I, or "records" in PASCAL. In the last two chapters we saw how to use macros to extend the control structures of Lisp; here we see how they can be used to extend Lisp's data structures as well.

19.1 Introduction to Structure Macros

To explain the basic idea, assume you were writing a Lisp program that dealt with space ships. In your program, you want to represent a space ship by a Lisp object of some kind. The interesting things about a space ship, as far as your program is concerned, are its position (X and Y), velocity (X and Y), and mass. How do you represent a space ship?

Well, the representation could be a list of the x-position, y-position, and so on. Equally well it could be an array of five elements, the zeroth being the x-position, the first being the y-position, and so on. The problem with both of these representations is that the "elements" (such as x-position) occupy places in the object which are quite arbitrary, and hard to remember (Hmm, was the mass the third or the fourth element of the array?). This would make programs harder to write and read. It would not be obvious when reading a program that an expression such as (cadddr ship1) or (aref ship2 3) means "the y component of the ship's velocity", and it would be very easy to write caddr in place of cadddr.

What we would like to see are names, easy to remember and to understand. If the symbol foo were bound to a representation of a space ship, then

```
(ship-x-position foo)
```

could return its x-position, and

```
(ship-y-position foo)
```

its y-position, and so forth. The defstruct facility does just this.

defstruct itself is a macro which defines a structure. For the space ship example above, we might define the structure by saying:

```
(defstruct (ship)
     ship-x-position
     ship-y-position
     ship-x-velocity
     ship-y-velocity
     ship-mass)
```

This says that every ship is an object with five named components. (This is a very simple case of defstruct; we will see the general form later.) The evaluation of this form does several things. First, it defines ship-x-position to be a function which, given a ship, returns the x component of its position. This is called an *accessor function*, because it *accesses* a component of a structure. defstruct defines the other four accessor functions analogously.

defstruct will also define **make-ship** to be a macro which expands into the necessary Lisp code to create a **ship** object. So **(setq x (make-ship))** will make a new ship, and set **x** to it. This macro is called the *constructor macro*, because it constructs a new structure.

We also want to be able to change the contents of a structure. To do this, we use the **setf** macro (see page 270), as follows (for example):

```
    (setf (ship-x-position x) 100)
```

Here **x** is bound to a ship, and after the evaluation of the **setf** form, the **ship-x-position** of that ship will be 100. Another way to change the contents of a structure is to use the alterant macro, which is described later, in section 19.4.3, page 310.

How does all this map into the familiar primitives of Lisp? In this simple example, we left the choice of implementation technique up to **defstruct**; it will choose to represent a ship as an array. The array has five elements, which are the five components of the ship. The accessor functions are defined thus:

```
    (defun ship-x-function (ship)
      (aref ship 0))
```

The constructor macro **(make-ship)** expands into **(make-array 5)**, which makes an array of the appropriate size to be a ship. Note that a program which uses ships need not contain any explicit knowledge that ships are represented as five-element arrays; this is kept hidden by **defstruct**.

The accessor functions are not actually ordinary functions; instead they are **substs** (see section 10.5.1, page 161). This difference has two implications: it allows **setf** to understand the accessor functions, and it allows the compiler to substitute the body of an accessor function directly into any function that uses it, making compiled programs that use **defstruct** exactly equal in efficiency to programs that "do it by hand." Thus writing **(ship-mass s)** is exactly equivalent to writing **(aref s 4)**, and writing **(setf (ship-mass s) m)** is exactly equivalent to writing **(aset m s 4)**, when the program is compiled. It is also possible to tell **defstruct** to implement the accessor functions as macros; this is not normally done in Zetalisp, however.

We can now use the **describe-defstruct** function to look at the **ship** object, and see what its contents are:

```
    (describe-defstruct x 'ship) =>

    #<art-q-5 17073131> is a ship
       ship-x-position:           100
       ship-y-position:           nil
       ship-x-velocity:           nil
       ship-y-velocity:           nil
       ship-mass:                 nil
    #<art-q-5 17073131>
```

(The **describe-defstruct** function is explained more fully on page 301.)

By itself, this simple example provides a powerful structure definition tool. But, in fact, **defstruct** has many other features. First of all, we might want to specify what kind of Lisp object to use for the "implementation" of the structure. The example above implemented a "ship" as an array, but **defstruct** can also implement structures as array-leaders, lists, and other things. (For array-leaders, the accessor functions call **array-leader**, for lists, **nth**, and so on.)

Most structures are implemented as arrays. Lists take slightly less storage, but elements near the end of a long list are slower to access. Array leaders allow you to have a homogeneous aggregate (the array) and a heterogeneous aggregate with named elements (the leader) tied together into one object.

defstruct allows you to specify to the constructor macro what the various elements of the structure should be initialized to. It also lets you give, in the defstruct form, default values for the initialization of each element.

The defstruct in Zetalisp also works in various dialects of Maclisp, and so it has some features that are not useful in Zetalisp. When possible, the Maclisp-specific features attempt to do something reasonable or harmless in Zetalisp, to make it easier to write code that will run equally well in Zetalisp and Maclisp. (Note that this defstruct is not necessarily the default one installed in Maclisp!)

19.2 How to Use Defstruct

defstruct *Macro*

A call to defstruct looks like:

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

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

Each *option* may be either a symbol, which should be one of the recognized option names listed in the next section, or a list, whose car should be one of the option names and the rest of which should be "arguments" to the option. Some options have arguments that default; others require that arguments be given explicitly.

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

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

Each *slot-description* allocates one element of the physical structure, even though in form (3) several slots are defined.

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

In form (1), *slot-name* simply defines a slot with the given name. An accessor function will be defined with the name *slot-name* (but see the :conc-name option, page 303). Form (2) is similar, but allows a default initialization for the slot. Initialization is explained further on page 308. Form (3) lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of **defstruct**, which is explained on page 310.

Because evaluation of a **defstruct** form causes many functions and macros to be defined, you must take care not to define the same name with two different **defstruct** forms. A name can only have one function definition at a time; if it is redefined, the latest definition is the one that takes effect, and the earlier definition is clobbered. (This is no different from the requirement that each **defun** which is intended to define a distinct function must have a distinct name.)

To systematize this necessary carefulness, as well as for clarity in the code, it is conventional to prefix the names of all of the accessor functions with some text unique to the structure. In the example above, all the names started with **ship-**. The **:conc-name** option can be used to provide such prefixes automatically (see page 303). Similarly, the conventional name for the constructor macro in the example above was **make-ship**, and the conventional name for the alterant macro (see section 19.4.3, page 310) was **alter-ship**.

The **describe-defstruct** function lets you examine an instance of a structure.

describe-defstruct *instance* &optional *name*

describe-defstruct takes an *instance* of a structure, and prints out a description of the instance, including the contents of each of its slots. *name* should be the name of the structure; you must provide the name of the structure so that **describe-defstruct** can know what structure *instance* is an instance of, and therefore figure out what the names of the slots of *instance* are.

If *instance* is a named structure, you don't have to provide *name*, since it is just the named structure symbol of *instance*. Normally the **describe** function (see page 641) will call **describe-defstruct** if it is asked to describe a named structure; however some named structures have their own idea of how to describe themselves. See page 312 for more information about named structures.

19.3 Options to Defstruct

This section explains each of the options that can be given to **defstruct**.

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

```
(defstruct (foo (:type :array)
                (:make-array (:type 'art-8b :leader-length 3))
                :conc-name
                (:size-symbol foo))
  a
  b)
```

:type The :type option specifies what kind of Lisp object will be used to implement the structure. It must be given one argument, which must be one of the symbols enumerated below, or a user-defined type. If the option itself is not provided, the type defaults to :array. You can define your own types; this is explained in section 19.9, page 316.

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

:named-array
Like :array, but make the array a named structure (see page 312) using the name of the structure as the named structure symbol. Element 0 of the array will hold the named structure symbol and so will not be used to hold a component of the structure.

:array-leader
Use an array, storing components in the leader of the array. (See the :make-array option, described below.)

:named-array-leader
Like :array-leader, but make the array a named structure (see page 312) using the name of the structure as the named structure symbol. Element 1 of the leader will hold the named structure symbol and so will not be used to hold a component of the structure.

:list Use a list.

:named-list
Like :list, but the first element of the list will hold the symbol that is the name of the structure and so will not be used as a component.

:fixnum-array
Like :array, but the type of the array is **art-32b**.

:flonum-array
Like :array, but the type of the array is **art-float**.

:tree The structure is implemented out of a binary tree of conses, with the leaves serving as the slots.

:fixnum
This unusual type implements the structure as a single fixnum. The structure may only have one slot. This is only useful with the byte field feature (see page 310); it lets you store a bunch of small numbers within fields of a fixnum, giving the fields names.

:grouped-array
This is described in section 19.6, page 312.

:constructor This option takes one argument, which specifies the name of the constructor macro. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string "make-" to the name of the structure. If the argument is provided and is **nil**, no constructor is defined. Use of the constructor macro is explained in section 19.4.1, page 308.

:alterant This option takes one argument, which specifies the name of the alterant macro. If the argument is not provided or if the option itself is not provided, the name of the alterant is made by concatenating the string "alter-" to the name of the structure. If the argument is provided and is **nil**, no alterant is defined. Use of the alterant macro is explained in section 19.4.3, page 310.

:predicate The :predicate option causes **defstruct** to generate a predicate to recognize instances of the structure. Naturally it only works for "named" types. The argument to the :predicate option is the name of the predicate. If the option is present without an argument, then the name is formed by concatenating "-p" to the end of the name symbol of the structure. If the option is not present, then no predicate is generated. Example:

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

defines a single argument function, foo-p, that is true only of instances of this structure.

:copier This option causes defstruct to generate a single argument function that will copy instances of this structure. Its argument is the name of the copying function. If the option is present without an argument, then the name is formed by concatenating "copy-" with the name of the structure. Example:

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

Generates a function approximately like:

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

:default-pointer

Normally, the accessors defined by defstruct expect to be given exactly one argument. However, if the :default-pointer argument is used, the argument to each accessor is optional. If you use an accessor in the usual way it will do the usual thing, but if you invoke it without its argument, it will behave as if you had invoked it on the result of evaluating the form which is the argument to the :default-pointer argument. Here is an example:

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

(room-name x) ==> (aref x 0)
(room-name)   ==> (aref *default-room* 0)
```

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

:conc-name It is conventional to begin the names of all the accessor functions of a structure with a specific prefix, usually the name of the structure followed by a hyphen. The :conc-name option allows you to specify this prefix and have it concatenated onto the front of all the slot names to make the names of the accessor functions. The argument should be a symbol; its print-name is used as the prefix. If :conc-name is specified without an argument, the prefix will be the name of the structure followed by a hyphen. If you do not specify the :conc-name option, the names of the accessors are the same as the slot names, and it is up to you to name the slots according to some suitable convention.

The constructor and alterant macros are given slot names, not accessor names. It is important to keep this in mind when using :conc-name, since it causes the slot and accessor names to be different. Here is an example:

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

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

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

:include This option is used for building a new structure definition as an extension of an old structure definition. Suppose you have a structure called person that looks like this:

```
(defstruct (person :conc-name)
   name
   age
   sex)
```

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

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

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

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

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

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

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

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

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

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

For example, if we had wanted to define **astronaut** so that the default age for an astronaut is 45., then we could have said:

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

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

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

The argument to the :make-array option should be a list of alternating keyword symbols to the **make-array** function (see page 126), and forms whose values are the arguments to those keywords. For example, (:make-array (:type 'art-16b)) would request that the type of the array be **art-16b**. Note that the keyword symbol is *not* evaluated.

defstruct overrides any of the :make-array options that it needs to. For example, if your structure is of type :array, then **defstruct** will supply the size of that array regardless of what you say in the :make-array option. If you use the :initial-value **make-array** option, it will initialize all the slots, but **defstruct's** own initializations will be done afterward.

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

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

:times
This option is used for structures of type :grouped-array to control the number of repetitions of the structure that will be allocated by the constructor macro. (See section 19.6, page 312.) The constructor macro will also allow :times to be used as a keyword that will override the value given in the original defstruct form. If :times appears in neither the invocation of the constructor nor in the :make-array option to defstruct, then the constructor will only allocate one instance of the structure.

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

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

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

:but-first
This option is best explained by example:

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

The accessors expand like this:

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

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

:callable-accessors
This option controls whether accessors are really functions, and therefore "callable", or whether they are really macros. With an argument of **t**, or with no argument, or if the option is not provided, then the accessors are really functions. Specifically, they are **substs**, so that they have all the efficiency of macros in compiled programs, while still being function objects that can be manipulated (passed to **mapcar**, etc.). If the argument is **nil** then the accessors will really be macros.

:eval-when Normally the functions and macros defined by **defstruct** are defined at eval-time, compile-time, and load-time. This option allows the user to control this behavior. The argument to the **:eval-when** option is just like the list that is the first subform of an **eval-when** special form (see page 231). For example, **(:eval-when (:eval :compile))** will cause the functions and macros to be defined only when the code is running interpreted or inside the compiler.

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

The **:property** option can be used to give a **defstruct** an arbitrary property. (**:property** *property-name value*) gives the **defstruct** a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user will have to look inside the **defstruct-description** structure himself (see page 315).

:print This allows the user to control the printed representation of his structure in an a way independent of the Lisp dialect in use. Here is an example:

```
(defstruct (foo :named
                (:print "#<Foo ~S ~S>"
                        (foo-a foo) (foo-b foo)))
  foo-a
  foo-b)
```

Of course, this only works if you use some named type, so that the system can recognize examples of this structure automatically.

The arguments to the **:print** option are arguments to the **format** function (except for the stream of course!). They are evaluated in an environment where the name symbol of the structure (**foo** in this case) is bound to the instance of the structure to be printed.

This works by defining, automatically, a named structure handler. Do not use the :print option if you define a named structure handler yourself, as they will conflict.

type In addition to the options listed above, any currently defined type (any legal argument to the :type option) can be used as an option. This is mostly for compatibility with the old version of defstruct. It allows you to say just *type* instead of (:type *type*). It is an error to give an argument to one of these options.

other Finally, if an option isn't found among those listed above, defstruct checks the property list of the name of the option to see if it has a non-nil :defstruct-option property. If it does have such a property, then if the option is of the form (*option-name value*), it is treated just like (:property *option-name value*). That is, the defstruct is given an *option-name* property of *value*. It is an error to use such an option without a value.

This provides a primitive way for you to define your own options to defstruct, particularly in connection with user-defined types (see section 19.9, page 316). Several of the options listed above are actually implemented using this mechanism.

19.4 Using the Constructor and Alterant Macros

After you have defined a new structure with defstruct, you can create instances of this structure using the constructor macro, and you can alter the values of its slots using the alterant macro. By default, defstruct defines both the constructor and the alterant, forming their names by concatenating "make-" and "alter-", respectively, onto the name of the structure. You can specify the names yourself by passing the name you want to use as the argument to the :constructor or :alterant options, or specify that you don't want the macro created at all by passing nil as the argument.

19.4.1 Constructor Macros

A call to a constructor macro, in general, has the form

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

Each *symbol* may be either the name of a *slot* of the structure, or a specially recognized keyword. All the *forms* are evaluated.

If *symbol* is the name of a *slot* (*not* the name of an accessor), then that element of the created structure will be initialized to the value of *form*. If no *symbol* is present for a given slot, then the slot will be initialized to the result of evaluating the default initialization form specified in the call to defstruct. (In other words, the initialization form specified to the constructor overrides the initialization form specified to defstruct.) If the defstruct itself also did not specify any initialization, the element's initial value is undefined. You should always specify the initialization, either in the defstruct or in the constructor macro, if you care about the initial value of the slot.

Notes: The order of evaluation of the initialization forms is *not* necessarily the same as the order in which they appear in the constructor call, nor the order in which they appear in the defstruct; you should make sure your code does not depend on the order of evaluation. The forms are re-evaluated on every constructor-macro call, so that if, for example, the form (gensym) were used as an initialization form, either in a call to a constructor macro or as a default initialization in the defstruct, then every call to the constructor macro would create a new symbol.

There are two symbols that are specially recognized by the constructor. They are :make-array, which should only be used for :array and :array-leader type structures (or the named versions of those types), and :times, which should only be used for :grouped-array type structures. If one of these symbols appears instead of a slot name, then it is interpreted just as the :make-array option or the :times option (see page 305), and it overrides what was requested in that option. For example:

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

19.4.2 By-position Constructor Macros

If the :constructor option is given as (:constructor *name arglist*), then instead of making a keyword driven constructor, defstruct defines a "function style" constructor, taking arguments whose meaning is determined by the argument's position rather than by a keyword. The *arglist* is used to describe what the arguments to the constructor will be. In the simplest case something like (:constructor make-foo (a b c)) defines make-foo to be a three-argument constructor macro whose arguments are used to initialize the slots named a, b, and c.

In addition, the keywords &optional, &rest, and &aux are recognized in the argument list. They work in the way you might expect, but there are a few fine points worthy of explanation:

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

This defines make-foo to be a constructor of one or more arguments. The first argument is used to initialize the a slot. The second argument is used to initialize the b slot. If there isn't any second argument, then the default value given in the body of the defstruct (if given) is used instead. The third argument is used to initialize the c slot. If there isn't any third argument, then the symbol sea is used instead. Any arguments following the third argument are collected into a list and used to initialize the d slot. If there are three or fewer arguments, then nil is placed in the d slot. The e slot *is not initialized*; its initial value is undefined. Finally, the f slot is initialized to contain the symbol eff.

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

Since there is so much freedom in defining constructors this way, it would be cruel to only allow the :constructor option to be given once. So, by special dispensation, you are allowed to give the :constructor option more than once, so that you can define several different constructors, each with a different syntax.

Note that even these "function style" constructors do not guarantee that their arguments will be evaluated in the order that you wrote them. Also note that you cannot specify the :make-array nor :times information in this form of constructor macro.

19.4.3 Alterant Macros

A call to the alterant macro, in general, has the form

```
(name-of-alterant-macro instance-form
     slot-name-1 form-1
     slot-name-2 form-2
     ...)
```

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

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

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

19.5 Byte Fields

The byte field feature of **defstruct** allows you to specify that several slots of your structure are bytes (see section 7.8, page 116) in an integer stored in one element of the structure. For example, suppose we had the following structure:

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

This will work correctly. However, it wastes space. Area codes and exchange numbers are always less than 1000., and so both can fit into 10. bit fields when expressed as binary numbers. Since Lisp Machine fixnums have (more than) 20. bits, both of these values can be packed into a single fixnum. To tell **defstruct** to do so, you can change the structure definition to the following:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  ((area-code #o1212 617.)
   (exchange #o0012))
  line-number)
```

The magic octal numbers #o1212 and #o0012 are byte specifiers to be used with the functions ldb and dpb. The accessors, constructor, and alterant will now operate as follows:

```
(area-code pbe) ==> (ldb #o1212 (caddr pbe))
(exchange pbe)  ==> (ldb #o0012 (caddr pbe))

(make-phone-book-entry
   name "Fred Derf"
   address "259 Octal St."
   exchange ex
   line-number 7788.)

==> (list "Fred Derf" "259 Octal St." (dpb ex 12 2322000) 17154)

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

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

Note that the alterant macro is optimized to only read and write the second element of the list once, even though you are altering two different byte fields within it. This is more efficient than using two setfs. Additional optimization by the alterant macro occurs if the byte specifiers in the defstruct slot descriptions are constants. However, you don't need to worry about the details of how the alterant macro does its work.

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

The byte specifier need not be a constant; a variable or, indeed, any Lisp form, is legal as a byte specifier. It will be evaluated each time the slot is accessed. Of course, unless you are doing something very strange you will not want the byte specifier to change between accesses.

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

1) Initializations for the entire word given in the defstruct form.

2) Initializations for the byte fields given in the defstruct form.

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

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

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

19.6 Grouped Arrays

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

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

Note that the "size" of the structure (for purposes of the :size-symbol and :size-macro options) is the number of elements in *one* instance of the structure; the actual length of the array is the product of the size of the structure and the number of instances. The number of instances to be created by the constructor macro is given as the argument to the :times option to defstruct, or the :times keyword of the constructor macro.

19.7 Named Structures

The *named structure* feature provides a very simple form of user-defined data type. Any array may be made a named structure, although usually the :named option of defstruct is used to create named structures. The principal advantages to a named structure are that it has a more informative printed representation than a normal array and that the describe function knows how to give a detailed description of it. (You don't have to use describe-defstruct, because describe can figure out what the names of the slots of the structure are by looking at the named structure's name.) Because of these improved user-interface features it is recommended that "system" data structures be implemented with named structures.

Another kind of user-defined data type, more advanced but less efficient when just used as a record structure, is provided by the *flavor* feature (see chapter 20, page 321).

A named structure has an associated symbol, called its "named structure symbol", which represents what user-defined type it is an instance of; the typep function, applied to the named structure, will return this symbol. If the array has a leader, then the symbol is found in element

1 of the leader; otherwise it is found in element 0 of the array. (Note: if a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be stored in any element of a numeric array.)

If you call **typep** with two arguments, the first being an instance of a named structure and the second being its named structure symbol, **typep** will return **t**. **t** will also be returned if the second argument is the named structure symbol of a **:named defstruct** included (using the **:include** option, see page 304), directly or indirectly, by the **defstruct** for this structure. For example, if the structure **astronaut** includes the structure **person**, and **person** is a named structure, then giving **typep** an instance of an **astronaut** as the first argument, and the symbol **person** as the second argument, will return **t**. This reflects the fact that an astronaut is, in fact, a person, as well as being an astronaut.

You may associate with a named structure a function that will handle various operations that can be done on the named structure. Currently, you can control how the named structure is printed, and what **describe** will do with it.

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

:which-operations
: Should return a list of the names of the operations the function handles.

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

  #<named-structure-symbol octal-address>

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

:sxhash
: The arguments are **:sxhash**, the named structure, and a flag. It should return a hash code to use as the value of **sxhash** for this structure. It is often useful to call **sxhash** on some (perhaps all) of the components of the structure and combine the results.

  The flag says that it is permissible to use the structure's address in forming the hash code. For some kinds of structure, there may be no way to generate a good hash code except to use the address. If the flag is **nil**, they must simply do the best they can, even if it means always returning zero.

It is permissible to return nil for :sxhash. Then sxhash will produce a hash code in its default fashion.

Here is an example of a simple named-structure handler function:

```
(defun (person named-structure-invoke) (op self &rest args)
  (selectq op
    (:which-operations '(:print-self))
    (:print-self
      (format (first args)
              (if (third args) "#<person ~A>" "~A")
              (person-name self)))))
```

For this definition to have any effect, the person defstruct used as an example earlier must be modified to include the :named attribute.

This handler causes a person structure to include its name in its printed representation; it also causes princ of a person to print just the name, with no "#<" syntax. Even though the astronaut structure of our examples :includes the person structure, this named-structure handler will not be invoked when an astronaut is printed, and an astronaut will not include his name in his printed representation. This is because named structures are not as general as flavors (see chapter 20, page 321).

The following functions operate on named structures.

named-structure-p *x*
: This semi-predicate returns nil if *x* is not a named structure; otherwise it returns *x*'s named structure symbol.

named-structure-symbol *x*
: *x* should be a named structure. This returns *x*'s named structure symbol: if *x* has an array leader, element 1 of the leader is returned, otherwise element 0 of the array is returned.

make-array-into-named-structure *array*
: *array* is made to be a named structure and is returned.

named-structure-invoke *operation structure* &rest *args*
: *operation* should be a keyword symbol, and *structure* should be a named structure. The handler function of the named structure symbol, found as the value of the named-structure-invoke property of the symbol, is called with appropriate arguments. (This function used to take its first two arguments in the opposite order, and that argument order will continue to work indefinitely, but it should not be used in new programs.)

See also the :named-structure-symbol keyword to make-array, page 126.

19.8 The si:defstruct-description Structure

This section discusses the internal structures used by **defstruct** that might be useful to programs that want to interface to **defstruct** nicely. For example, if you want to write a program that examines structures and displays them the way **describe** (see page 641) and the Inspector do, your program will work by examining these structures. The information in this section is also necessary for anyone who is thinking of defining his own structure types.

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

The **si:defstruct-description** structure is defined as follows, in the **system-internals** package (also called the **si** package):

```
(defstruct (defstruct-description
             (:default-pointer description)
             (:conc-name defstruct-description-))
           name
           size
           property-alist
           slot-alist)
```
(This is a simplified version of the real definition. There are other slots in the structure, which we aren't telling you about.)

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

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

The **property-alist** slot contains an alist with pairs of the form (*property-name . property*) containing properties placed there by the **:property** option to **defstruct** or by property names used as options to **defstruct** (see the **:property** option, page 307).

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

```
(defstruct (defstruct-slot-description
                (:default-pointer slot-description)
                (:conc-name defstruct-slot-description-))
           number
           ppss
           init-code
           ref-macro-name)
```
(This is also a simplified version of the real definition.)

The number slot contains the number of the location of this slot in an instance of the structure. Locations are numbered, starting with 0, and continuing up to a number one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure; see page 317.

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

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

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

19.9 Extensions to Defstruct

The macro defstruct-define-type can be used to teach defstruct about new types that it can use to implement structures.

defstruct-define-type *Macro*
This macro is used for teaching defstruct about new types; it is described in the rest of this chapter.

19.9.1 An Example

Let us start by examining a sample call to defstruct-define-type. This is how the :list type of structure might have been defined:

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

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

The keyword :cons is followed by a list of three variables that will be bound while the constructor-creating form is evaluated. The first, initialization-list, will be bound to a list of the initialization forms for the slots of the structure. The second, description, will be bound to the defstruct-description structure for the structure (see page 315). The third variable and the :list keyword will be explained later.

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

19.9.2 Syntax of defstruct-define-type

The syntax of defstruct-define-type is:

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

where each *option* is either the symbolic name of an option or a list of the form (*option-name* . *rest*). Different options interpret *rest* in different ways. The symbol *type* is given an si:defstruct-type-description property of a structure that describes the type completely.

19.9.3 Options to defstruct-define-type

This section is a catalog of all the options currently known about by defstruct-define-type.

:cons With the :cons option to defstruct-define-type you supply defstruct with the code to cons up a form that will construct an instance of a structure of this type.

The :cons option has the syntax:

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

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

The symbol *inits* will be bound to the information that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization. There is the :list kind, where *inits* is bound to a list of initializations, in the correct order, with nils in uninitialized slots. And there is the :alist kind, where *inits* is bound to an alist with pairs of the form (*slot-number* . *init-code*).

The symbol *description* will be bound to the instance of the defstruct-description structure (see page 315) that defstruct maintains for this particular structure. This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* will be bound to an alist with pairs of the form (*keyword* . *value*), where each *keyword* was a keyword supplied to the constructor macro that wasn't the name of a slot, and *value* was the Lisp object that followed the keyword. This is how you can make your own special keywords, like the existing :make-array and :times keywords. See the section on using the constructor macro, section 19.4.1, page 308. You specify the list of acceptable keywords with the :keywords option (see page 319).

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

:ref With the :ref option to **defstruct-define-type** you supply **defstruct** with the code to cons up a form that will reference an instance of a structure of this type.

The :ref option has the syntax:

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

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

The symbol *number* will be bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the **defstruct-slot-description** structure (see page 316).

The symbol *description* will be bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that will be defaulted by the :default-pointer option (see page 303). **defstruct** will check that the user has supplied exactly *n* arguments to the accessor function before calling the reference consing code.

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

:overhead The :overhead option to **defstruct-define-type** is how the user declares to **defstruct** that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

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

This number is used only by the :size-macro and :size-symbol options to **defstruct** (see page 306).

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

:keywords The :keywords option to defstruct-define-type allows the user to define additional constructor keywords for this type of structure. (The :make-array constructor keyword for structures of type :array is an example.) The syntax is: (:keywords *keyword-1* ... *keyword-n*) where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist (explained above).

:predicate With the :predicate option to defstruct-define-type, defstruct is told how to produce predicates for a particular type (for the :predicate option to defstruct). Its syntax is:

```
(predicate (description name)
  body...)
```

The variable *description* will be bound to the defstruct-description structure maintained for the structure we are to generate a predicate for. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code to evaluate to return the defining form for the predicate. A typical use of this option might look like:

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

:copier defstruct knows how to generate a copier function using the constructor and reference macro code that must be provided with any new defstruct type. Nevertheless it is sometimes desirable to specify a specific method of copying a particular defstruct type. The :copier option to defstruct-define-type allow this to be done:

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

As with the :predicate option, *description* is bound to an instance of the defstruct-description structure, *name* is bound to the symbol to be defined, and *body* is some code to evaluate to get the defining form. For example:

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

:defstruct The :defstruct option to defstruct-define-type allows the user to run some code and return some forms as part of the expansion of the defstruct macro.

The :defstruct option has the syntax:

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

body is a piece of code that will be run whenever defstruct is expanding a defstruct form that defines a structure of this type. The symbol *description* will be bound to the instance of the defstruct-description structure that defstruct maintains for this particular structure.

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