20. 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.

20.1 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)
  "Represents a space 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 also defines make-ship to be a macro or function (you can specify which one) that can create a ship object. So (setq s (make-ship)) makes a new ship, and sets s to it. This is called the *constructor*, 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 36), as follows (for example):

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

Here s is bound to a ship, and after the evaluation of the setf form, the ship-x-position of that ship is 100. Another way to change the contents of a structure is to use the alterant macro, which is described later, in section 20.4.3, page 387.

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; by default, it chooses 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-position (ship)
  (aref ship 0))
```

The constructor form (make-ship) performs (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 11.5.1, page 230). 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 (setf (aref s 4) m), 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 376.)

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. Packages are this sort of an object, and so are the strings which Zmacs uses for storing lines of text.

The constructor function or macro allows you to specify values for slots in the new structure. defstruct allows you to specify default initial values for slots; whenever a structure is constructed and no value is specified for a slot, the slot's default initial value is stored in it.

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 one installed in Maclisp!)

Note that there is another version of defstruct used in Common Lisp programs, which is slightly incompatible. See section 20.8, page 393.

20.2 How to Use Defstruct

defstruct *Macro*

A call to defstruct looks like:

```
(defstruct (name options...)
  [doc-string]
  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.

doc-string is a string which is recorded as the documentation of *name* as a structure. It can be accessed via (documentation '*name* 'structure). It is not required.

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

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

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 is defined with the name *slot-name* (but see the :conc-name option, page 379). Form (2) is similar, but allows a default initialization for the slot. Initialization is explained further on page 385. 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 387.

Forms (2) and (3) allow *slot-options* which are alternating keywords and values (unevaluated). These slot option keywords are defined:

:read-only *flag* If *flag* is non-nil, this specifies that this slot should not be changed in an existing structure. setf will not be allowed on the slot's accessor.

:type *type-spec* Declares that the contents of this slot must be of type *type-spec*. The Lisp machine compiler does not use this information, but sometimes it enables defstruct to deduce that it can pack the structure into less space by using a specialized array type.

:documentation *documentation-string*
Makes *documentation-string* the documentation for the slot's accessor function. It also goes in the si:defstruct-slot-description-documentation for this slot in the defstruct-description structure.

Here is an eggsample of using slot options:

```
(defstruct (eggsample :named :conc-name)
  (yolk 'a
   :documentation "First thing you need in an eggsample.")
  (grade 3)
  (albumen nil :read-only t))

(documentation 'eggsample-yolk 'function)
=> "First thing you need in an eggsample."

(setf (eggsample-albumen (make-eggsample)) 'eggsistential)
>>ERROR: SETF is forbidden on EGGSAMPLE-ALBUMEN.
While in the function ...
```

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 379). Similarly, the conventional name for the constructor in the example above was make-ship, and the conventional name for the alterant macro (see section 20.4.3, page 387) 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 791) calls 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 390 for more information about named structures.

20.3 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 (mod 256)))
                (:make-array (:leader-length 3))
                :conc-name
                (:size-macro foo))
  a b)
```

:type

The :type option specifies what kind of Lisp object to use 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 in traditional programs, or :vector in Common Lisp programs. You can define your own types; this is explained in section 20.10, page 396.

:list Uses a list.

:named-list

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

:array
:typed-array
:vector

These are all synonymous. They use an array, storing components in the body of the array.

:named-array

Like :array, but makes the array a named structure (see page 390) using the name of the structure as the named structure symbol. Element 0 of

the array holds the named structure symbol and so is not used to hold a component of the structure.

:named-typed-array
:named-vector
These two synonyms are like :named-array but the array always has a leader and the named structure symbol is stored there. As a result, it is possible to use the :subtype option to specify a restricted array type, such as art-8b.

:phony-named-vector
This is what you get in Common Lisp if you say (:type :vector) and :named.

: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 makes the array a named structure (see page 390) using the name of the structure as the named structure symbol. Element 1 of the leader holds the named structure symbol and so is not used to hold a component of the structure.

: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.

:named-fixnum-array
:named-flonum-array
Like :fixnum-array or :flonum-array but also a named structure, with a leader to hold the named structure symbol.

: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 387); 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 20.6, page 389.

The argument of :type may also have the form (*type subtype*). This is equivalent to specifying *type* for the :type option and *subtype* for the :subtype option. For example, (:type (:array (mod 16.))) specifies an array of four-bit bytes.

:subtype For structures which are arrays, :subtype permits the array type to be specified. It requires an argument, which must be either an array type name such as art-4b or a type specifier restricting the elements of the array. In other words, it should

be a suitable value for either the *type* or the *element-type* argument to make-array.

If no :subtype option is specified but a :type slot option is given for every slot, defstruct may deduce a subtype automatically to make the structure more compact.

See section 2.3, page 14 for more information on type specifiers.

:constructor Specifies how to make a constructor for the structure. In the simplest use, there is one argument, which specifies the name to give to the standard keyword-argument constructor. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string "make-" to the name of the structure. If the argument is provided and is nil, no constructor is defined. More complicated usage is explained in section 20.4.1, page 385.

:alterant Takes one argument, which specifies the name of the alterant macro. If the argument 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 20.4.3, page 387.

In Common Lisp programs, the default for :alterant is nil; no alterant is defined. In traditional programs, the default is alter-*name*.

:predicate 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)
  a
  b)
```

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

The defaulting of the :predicate option is different (and complicated) in Common Lisp programs. See section 20.8, page 393.

:copier Causes defstruct to generate a single argument function that can 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 the accessor is used with no argument, it evaluates the default-pointer form to find a structure and accesses the appropriate component of that structure. 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 string to be used as the prefix, or a symbol whose pname is to be used. If :conc-name is specified without an argument, the prefix is the name of the structure followed by a hyphen. If the argument is nil or "", 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.

In Common Lisp programs, the default for :conc-name, when this option is not specified, is the structure name followed by a hyphen. For traditional programs, the default is nil.

The keywords recognized by the constructor and alterant are the slot names, not the accessor names, transfered into the keyword package. 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 Builds 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 :named :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 :named (:include person)
                      :conc-name)
   helmet-size
   (favorite-beverage 'tang))
```

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 :include option inserts the slots of the included structure at the front of the list of slots for this structure. That is, an astronaut has 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, such as person-name, can be used also on astronaut's. New accessor functions are generated for these slots in the astronaut structure as if they were defined afresh; their names start with astronaut- instead of person-. In fact, the functions person-age and astronaut-age receive identical definitions.

Since the structures are named structures, recognizable by typep, subtypep considers astronaut a subtype of person, and typep considers any astronaut to be of type person.

The following examples illustrate how you can construct and use astronaut structures:

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

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

(typep x 'astronaut) => t
(typep x 'person) => t
```

Note that the :conc-name option was *not* inherited from the included structure; it is present for :astronaut only because it was specified explicitly in the definition. Similarly, the :default-pointer and :but-first options are not inherited from the :include'd structure.

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

(:include *name new-descriptor-1 ... new-descriptor-n*)

Each *new-slot-descriptor* is just like the slot descriptors used for defining new slots, except that byte fields are not allowed. The default initialization specified in *new-slot-descriptor*, or the absence of one, overrides what was specified in the included structure type (person). Any slot option values specified in *new-slot-descriptor* also override the values given in the included structure's definition. Any inherited slots for which no *new-slot-descriptor* is given, and any slot options not explicitly overridden, are inherited.

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

```
(defstruct (astronaut :conc-name
                      (:include person
                        (age 45. :documentation
                          "The ASTRONAUT's age in years.")))
   helmet-size
   (favorite-beverage 'tang))
```

If the :read-only option is specified as nil when t would have been inherited, an error is signaled.

: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 is used instead. Asking for type :array and giving the :named option as well is the same as asking for the type :named-array; the only difference is stylistic.

The :named option works quite differently in Common Lisp programs; see section 20.8, page 393.

: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.

The argument to the :make-array option should be a list of alternating keyword symbols for the make-array function (see page 167), and forms whose values are the arguments to those keywords. For example, (:make-array (:area 'permanent-storage-area)) would request that the array be allocated in a particular area. 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 supplies the size of that array regardless of what you say in the :make-array option. If you use the :initial-element make-array option, it initializes all the slots, but defstruct's own initializations are done afterward. If a subtype has been specified to or deduced by defstruct, it overrides any :type keyword in the :make-array argument.

Constructors for structures implemented as arrays recognize the keyword argument :make-array. 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 chooses appropriate defaults. The :make-array option may only be used with the default style of constructor that takes keyword arguments.

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 Used for structures of type :grouped-array to control the number of repetitions of the structure to be allocated by the constructor. (See section 20.6, page 389.) The constructor also accepts a keyword argument :times to override the value given in the defstruct. If :times appears in neither the invocation of the constructor nor as a defstruct option, the constructor allocates only one instance of the structure.

:size-macro Defines a special macro to expand into the size of this structure. 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 (for example, the length of the array or list). The argument of the :size-macro option is the name to be used for the macro. If this option is present without an argument, then the name of the structure is concatenated with '-size' to produce the macro name.

Example:

```
(defstruct (foo :conc-name :size-macro)
  a b)
(macroexpand '(foo-size)) => 2
```

:size-symbol Like :size-macro but defines a global variable rather than a macro. The size of the structure is the variable's value. Use of :size-macro is considered cleaner.

: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 is 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

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 are really macros.

:callable-constructors

Controls whether constructors are really functions, and therefore "callable", or macros. An argument of t makes them functions; nil makes them macros. The default is t in Common Lisp programs, nil in traditional programs. See section 20.4.1, page 385 for more information.

: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 must look inside the defstruct-description structure himself (see page 394).

:print Controls the printed representation of his structure in 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 generating a defselect that creates a named structure handler. Do not use the :print option if you define a named structure handler yourself, as they will conflict.

:print-function

is the Common Lisp version of the :print option. Its argument is a function to print a structure of this type, and it is called with three arguments: the structure to be printed, the stream to print it on, and the current printing depth (which should be compared with *print-level* to decide when to cut off recursion and print '#'). The function is expected to observe the values of the various printer-control variables such as *print-escape* (see page 514). Example:

```
(defstruct (bar :named :conc-name
                (:print-function
                  (lambda (struct stream depth)
                    depth ;unused
                    (sys:printing-random-object
                        (struct stream :type)
                      (format stream "with zap ~S"
                              (bar-zap struct))))))
  "The famous BAR structure."
  (zap 'yow)
  random-slot)

(make-bar) => #<BAR with zap YOW>
```

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, it should be a valid defstruct-keyword for the type of structure being defined, and the option should

be of the form (*option-name value*). If so, the option is treated just like (:property *option-name value*). That is, the defstruct is given an *option-name* property of *value*.

This provides a primitive way for you to define your own options to defstruct, particularly in connection with user-defined types (see section 20.10, page 396). Several of the options listed above are actually implemented using this mechanism. They include :times, :subtype and :make-array.

The valid defstruct-keywords of a type are in a list in the defstruct-keywords slot of the defstruct-type-description structure for *type*.

20.4 Using the Constructor and Alterant

After you have defined a new structure with defstruct, you can create instances of this structure using the constructor, and you can alter the values of its slots using the alterant macro. By default, traditional defstruct defines both the constructor and the alterant, forming their names by concatenating 'make-' and 'alter-', respectively, onto the name of the structure. The defstruct for Common Lisp programs defines no alterant by default. 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.

20.4.1 Constructors

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

(*name-of-constructor keyword-1 value-1 keyword-2 value-2 ...*)

Each *keyword* is a keyword (a symbol in the keyword package) whose name matches one of the slots of the structure, or one of a few specially recognized keywords.

The name of the constructor is specified by the :constructor option, which can also specify a documentation string for it:

(:constructor *name-of-constructor* [*doc-string*])

If a *keyword* matches the name of a slot (*not* the name of an accessor), then the corresponding *value* is used to initialize that slot of the new structure. Any slots whose values are not specified in this way are initialized to the values of the default initial value forms specified in the defstruct. If no default initial value was specified either for a slot, that slot's initial value is undefined. You should always specify the initialization, either in the defstruct or in the constructor invocation, if you care about the initial value of the slot.

Constructors may be macros or functions. They are functions if the :callable-constructors option to defstruct is non-nil. By default, they are functions in Common Lisp programs and macros in traditional programs.

Constructor macros allow the slot name (in its own package) to be used instead of a keyword. Constructor functions do not, as they are ordinary functions defined using &key. Old code using slot names not in the keyword package should be converted.

The default initial value forms are evaluated (if needed) each time a structure is constructed, so that if (gensym) is used as a default initial value form then a new symbol is generated for each structure. The order of evaluation of the default initial value forms is unpredictable. When the constructor is a macro, the order of evaluation of the keyword argument forms it is given is also unpredictable.

The two special keyword arguments recognized by constructors are :make-array and :times. :make-array should be used only for structures which are represented as arrays, and :times only for :grouped-array structures. If one of these arguments is given, then it overrides the :make-array option or the :times option (see page 381) specified in the defstruct. For example:
For example,

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

User-defined types of structures can define their own special constructor keywords.

20.4.2 By-Position Constructors

If the :constructor option is given as (:constructor *name arglist* [*doc-string*]), then instead of making a keyword driven constructor, defstruct defines a positional constructor, taking arguments whose meaning is determined by the argument's position rather than by a keyword. The *arglist* is used to describe what arguments the constructor should accept. 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))
  "Make a FOO, with positional arguments")
```

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, even if a default value was specified in its slot-description. 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 override completely 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. These may include both keyword and positional constructors. If there are multiple keyword constructors, they all behave the same, differing only in the name. It is important to have a keyword constructor because otherwise the #S reader construct cannot work.

Note that positional constructors may be macros or functions, just like keyword constructors, and based on the same criterion: they are functions if the :callable-constructors option to defstruct is non-nil. By default, they are functions in Common Lisp programs and macros in traditional programs. If the positional constructor is a macro, then the actual order of evaluation of its arguments is unpredictable.

Also note that you cannot specify the :make-array or :times information in a positional constructor.

20.4.3 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 constructor macros, 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.

20.5 Byte Fields

The byte field feature of defstruct allows you to specify that several slots of your structure are bytes (see section 7.9, page 155) 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 works correctly but 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 (byte 10. 10.) 617.)
   (exchange (byte 10. 0)))
  line-number)
```

The expressions (byte ...) calculate 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 (byte 10. 10.) (caddr pbe))
(exchange pbe)  ==> (ldb (byte 10. 0) (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 (byte 10. 0) 631808.)
          7788.)

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

==> ((lambda (#:g0530)
       (setf (nth 2 #:g0530)
             (dpb ac (byte 10. 10.)
                     (dpb ex (byte 10. 0) (nth 2 #:g0530)))))
     pbe)
```

(This is how the expression would print; this text would not read in properly because a new uninterned symbol would be created by each use of #:.)

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 setf's. 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 is 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 is 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.

Constructors (both functions and 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 invocation.

4) Initializations for the byte fields given in the constructor invocation.

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 is unpredictable.

20.6 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 is taken from the :times keyword of the constructor or the argument to the :times option to defstruct.

20.7 Named Structures

The *named structure* feature provides a very simple form of user-defined data type. (Flavors are another, more powerful, facility for defining data types, but they are more expensive in simple cases. See chapter 21, page 401.) A named structure is actually an array, containing elements and optionally a leader. The difference between a named structure and an ordinary array is that the named structure also contains an explicit slot to hold its ostensible data type. This data type is a symbol, any symbol the programmer cares to use. In traditional programs, named structures are normally defined using **defstruct** with the **:named** option. In Common Lisp programs, **defstruct** defines a named structure by default. Individual named structures are made with the constructors defined by **defstruct**.

The data type symbol of a named structure is also called the *named structure symbol*. It is stored in array element 0 if the structure has no leader. If there is a leader, the type symbol is stored in array leader element 1 (recall that element 0 is reserved for the fill pointer). If a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be an element of a numeric array.

Named structure are *recognizable*; that is, if you define a named structure called **foo**, you can always tell whether an object is a **foo** structure. No array created in the normal fashion, no matter what components it has, will be mistaken for a genuine **foo**.

Named structures can recognized by **typep**. Specify **foo**, the named structure name, as the second argument, and the object to be tested as the first argument. **type-of** of an ordinary array returns **array**, but **type-of** of a named structure returns the explicitly recorded data type symbol.

```
(defstruct (foo :named) a b)
(type-of (make-foo)) => foo
(typep (make-foo) 'foo) => t
```

Named structures of other types which include **foo** are also recognized as **foo**'s by **typep**. For example, using the previously-given definitions of **person** and **astronaut**, then

```
(typep (make-astronaut) 'person) => t
```

because the type **person** was explicitly included by the **defstruct** for **astronaut**. Indirect includes count also:

```
(defstruct (mission-specialist :named
             (:include astronaut))
  ...)

(typep (make-mission-specialist) 'person) => t
(subtypep 'person 'mission-specialist) => t
```

It should be emphasized that the named structure *is* an array. All the usual array functions, such as **aref** and **array-dimension**, can be used on it. If it is one-dimensional (as is usually the case) then the named structure is a vector and the generic sequence functions can be used on it.

```
(typep (make-astronaut) 'array) => t
(arrayp (make-astronaut)) => t
(array-rank (make-astronaut)) => 1
```

Because named structure data types are recognizable, they can define generic operations and say how to handle them. A few such operations are defined by the system and are invoked automatically from well-defined places. For example **print** automatically invokes the **:print-self** operation if you give it a named structure. Thus, each type of named structure can define how it should print. The standardly defined named structure operations are listed below. You can also define new named structure operations and invoke them by calling the named structure as a function just as you would invoke a flavor instance.

Operations on a named structure are all handled by a single function, which is found as the **named-structure-invoke** property of the structure type symbol. It is OK for a named structure type to have no handler function. Then invocation of any operation on the named structure returns **nil**, and system routines such as **print** take default actions.

If a handler function exists, it is given these arguments:

operation The name of the operation being invoked; usually a keyword.

structure The named structure which is being operated on.

additional-arguments...
Any other arguments which were passed when the operation was invoked. The handler function should have a rest parameter so it can accept any number of arguments.

The handler function should return **nil** if it does not recognize the *operation*. These are the named structure operations used by the system at present:

:which-operations
Should return a list of the names of the operations the function handles. Every handler function must handle this operation, and every opertation that the function handles should be included in this list.

:print-self Should output the printed representation of the named structure to a stream. The additional arguments are the stream to output to, the current depth in list-structure, and the current value of ***print-escape***. If **:print-self** is not in the value returned by **:which-operations**, or if there is no handler function, **print** uses **#s** syntax.

:describe Is invoked by **describe** and should output a description of the structure to ***standard-output***. If there is no handler function or **:describe** is not in its **:which-operations** list, **describe** prints the names and values of the structure's fields as defined in the **defstruct**.

:sxhash Is invoked by **sxhash** and 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 in some way.

There is one additional argument to this operation: a flag saying whether 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 produces a hash code in its default fashion.

:fasd-fixup Is invoked by fasload on a named structure that has been created from data in a QFASL file. The purpose of the operation is to give the structure a chance to "clean itself up" if, in order to be valid, it needs to have contents that are not exactly identical to those that were dumped. For example, readtables push themselves onto the list si:*all-readtables* so that they can be found by name.

For most kinds of structures it is acceptable not to define this operation at all (so that it returns nil).

Example handler function:

```
(defun (:property person named-structure-invoke)
       (op self &rest args)
  (selectq op
    (:which-operations '(:print-self :describe))
    (:describe
      (format (car args)
              "This is a ~D-year-old person"
              (person-age self)))
    (:print-self
      (if *print-escape*
          (si:printing-random-object (self (car args) :type)
            (princ (person-name self) (car args)))
        (princ (person-name self) (car args))))))
```

or

```
(defselect ((:property person named-structure-handler)
            ignore)
  (:print-self (self stream ignore &optional ignore)
    (if *print-escape*
        (si:printing-random-object (self stream :type)
          (princ (person-name self) stream))
      (princ (person-name self) stream)))
  (:describe (self)
    (format *standard-output*
            "This is a ~D-year-old person"
            (person-age self))))
```

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. This simple example could have been done even more simply with the :print-function option.

It is often convenient to define a handler function with defselect; but you must be careful. defselect by default defines the function to signal an error if it is called with a first argument that is not recognized. A handler function should return nil and get no error. To avoid the problem, specify ignore as the default handler when you write the defselect. See page 236.

Note that the handler function of a named structure type is *not* inherited by other named structure types that include it. For example, the above definition of a handler for person has no effect at all on the astronaut structure. If you need such inheritance, you must use flavors rather than named structures (see chapter 21, page 401).

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.

make-array-into-named-structure *array*

Marks *array* as a named structure and returns it. This is used by make-array when creating named structures. You should not normally call it explicitly.

named-structure-invoke *operation structure* &rest *args*

Invokes a named structure operation on *structure*. *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.

If the structure type has no named-structure-invoke property, nil is returned. By convention, nil is also returned by the handler if it does not recognize operation.

(send *structure operation args...*) has the same effect, by calling named-structure-invoke.

See also the :named-structure-symbol keyword to make-array, page 167.

20.8 Common Lisp Defstruct

cli:defstruct *Macro*

The version of defstruct used in Common Lisp programs differs from the traditional defstruct in the defaults for a few options and the meanings of a few of them.

The :conc-name option defaults to the structure type name followed by a hyphen in cli:defstruct. In traditional defstruct it defaults to nil.

The :callable-constructors option defaults to t in cli:defstruct, so that the constructor is a function. Traditionally, it defaults to nil.

The :alterant option defaults to nil in cli:defstruct, so that no alterant is defined. Traditionally, an alterant is defined by default with the name alter-*name*.

The :type option defaults to :named-vector in cli:defstruct. This makes a named structure, and you may specify how to print it. The :predicate option defaults to t in this case.

If the :type option is specified in cli:defstruct, you never get a named structure. You get either a plain list or a plain vector. There is no type-testing predicate, and you may not request one. You may not say how to print the structure, either.

If you specify the :named option along with :type, you still *do not* get a named structure. You get a plain list or a plain vector in which the structure name happens to be stored. The type is either :named-list or :phony-named-vector. The :predicate option defaults to nil, but you may specify t yourself. However, any randomly created list or vector with the structure name stored in the right place will satisfy the predicate thus defined. typep cannot recognize these phony named structures, and you may not specify how to print them (they do not understand named-structure-invoke.)

20.9 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 791) and the Inspector do, your program should 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 can be found as the si:defstruct-description property of the name of the structure; it contains such useful information as the number of slots in the structure, the defstruct options specified, and so on.

This is a simplified version of the way the si:defstruct-description structure is defined. It omits some slots whose meaning is not worth documenting here. (The actual definition is in the system-internals package.)

```
(defstruct (defstruct-description
             (:default-pointer description)
             (:conc-name defstruct-description-))
           name
           size
           property-alist
           slot-alist
           documentation)
```

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 slots 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 383).

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 (with other slots that are omitted here), also in the si package:

```
(defstruct (defstruct-slot-description
                (:default-pointer slot-description)
                :conc-name)
  number
  ppss
  init-code
  type
  property-alist
  ref-macro-name
  documentation)
```

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 397.

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 a canonical object which can be obtained (for comparison using eq) as the result of (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).

20.10 Extensions to Defstruct

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

Macro

si:defstruct-define-type

Is used for teaching defstruct about new types.

The syntax of si:defstruct-define-type is:

```
(si: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.

The semantics of si:defstruct-define-type is the subject of the rest of this section.

20.10.1 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:

```
(si: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 394). 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.

20.10.2 Options to si:defstruct-define-type

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

:cons Specifies 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*). Additional kinds may be provided in the future.

The symbol *description* will be bound to the instance of the defstruct-description structure (see page 394) 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 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, section 20.4.1, page 385. You specify the list of acceptable keywords with the :cons-keywords option (see page 398).

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

:ref Specifies 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 395).

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

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that will be defaulted by the :default-pointer option (see page 379). 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 si:defstruct-define-type.

:overhead Declares to defstruct that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location. For example, named arrays have an overhead of one, and named array leaders an overhead of two, but named typed arrays have no overhead since the structure type symbol is stored in the array leader whilst the actual data specifying the values of the slots is stored in the array proper.

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 382).

:named Controls the use of the :named option to defstruct. With no argument, the :named option means that this type is one which records the structure type name somehow (not necessarily by using an actual 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 the user specifies this type and :named as well. For example, the definition of the :list type contains (:named :named-list), saying that a defstruct that specifies (:list :named) really uses type :named-list.

:cons-keywords

Defines additional constructor keywords for this type of structure. Using these keywords, one may specify additional information about a structure at the time it is created ("consed") using one of its constructor functions or macros. (The :times constructor keyword for structures of type :grouped-array is an example.) The syntax is: (:cons-keywords *keyword-1 ... keyword-n*) where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist (explained above).

:keywords :keywords is an old name for the :cons-keywords option.

:defstruct-keywords

Defines additional defstruct options allowed for this type of structure. (The :subtype option for structures of type :array is an example.) These options take effect at the time the structure is defined using defstruct, and thus affect all structures of a particular type (unless overridden in some way.) In contrast, the :cons-keywords options affect the creation of individual structures of a particular

type.

The syntax is: (:defstruct-keywords *keyword-1 ... keyword-n*) where each *keyword* is a keyword that defstruct will recognize as an option. defstruct puts such options, with their values, in the property-alist slot of the defstuct-description structure (defined above)

:predicate Tells defstruct how to produce predicates for a particular type (for the :predicate option to defstruct). Its syntax is:

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

The variable *description* is bound to the defstruct-description structure maintained for the structure for which a predicate is being generated. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code to compute 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)
              ',(si:defstruct-description-name
                  description)))))
```

:copier defstruct knows how to generate a copier function using the constructor and reference 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 si: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)
  '(fdefine ',name 'copy-frobboz))
```

:defstruct The :defstruct option to si: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.

defstruct will cause the *body* forms to be evaluated as early as possible in the parsing of a structure definition, and cause the returned forms to be evalutated as late as possible in the macro-expansion of the defstuct forms. This is so that *body* can rehack arguments, signal errors, and the like before many of defstruct's internal forms are executed, while enabling it to return code which will modify or extend the default forms produced by a vanilla defstruct.