Version 15.17, X3J13/94-101.
Wed 11-May-1994 6:57pm EDT

Programming Language—Common Lisp

8. Structures

defstruct *Macro*

Syntax:

defstruct *name-and-options* [*documentation*] {↓*slot-description*}*
→ *structure-name*

name-and-options::=*structure-name* | (*structure-name* ⟦↓*options*⟧)

options::=↓*conc-name-option* |
{↓*constructor-option*}* |
↓*copier-option* |
↓*include-option* |
↓*initial-offset-option* |
↓*named-option* |
↓*predicate-option* |
↓*printer-option* |
↓*type-option*

conc-name-option::=`:conc-name` | (`:conc-name`) | (`:conc-name` *conc-name*)

constructor-option::=`:constructor` |
(`:constructor`) |
(`:constructor` *constructor-name*) |
(`:constructor` *constructor-name constructor-arglist*)

copier-option::=`:copier` | (`:copier`) | (`:copier` *copier-name*)

predicate-option::=`:predicate` | (`:predicate`) | (`:predicate` *predicate-name*)

include-option::=(`:include` *included-structure-name* {↓*slot-description*}*)

printer-option::=↓*print-object-option* | ↓*print-function-option*

print-object-option::=(`:print-object` *printer-name*) | (`:print-object`)

print-function-option::=(`:print-function` *printer-name*) | (`:print-function`)

type-option::=(`:type` *type*)

named-option::=`:named`

initial-offset-option::=(`:initial-offset` *initial-offset*)

defstruct

slot-description::=slot-name |
(*slot-name* [*slot-initform* ⟦↓*slot-option*⟧])

slot-option::=`:type` *slot-type* |
`:read-only` *slot-read-only-p*

Arguments and Values:

conc-name—a *string designator*.

constructor-arglist—a *boa lambda list*.

constructor-name—a *symbol*.

copier-name—a *symbol*.

included-structure-name—an already-defined *structure name*. Note that a *derived type* is not permissible, even if it would expand into a *structure name*.

initial-offset—a non-negative *integer*.

predicate-name—a *symbol*.

printer-name—a *function name* or a *lambda expression*.

slot-name—a *symbol*.

slot-initform—a *form*.

slot-read-only-p—a *generalized boolean*.

structure-name—a *symbol*.

type—one of the *type specifiers* **list**, **vector**, or `(vector` *size*`)`, or some other *type specifier* defined by the *implementation* to be appropriate.

documentation—a *string*; not evaluated.

Description:

defstruct defines a structured *type*, named *structure-type*, with named slots as specified by the *slot-options*.

defstruct defines *readers* for the slots and arranges for **setf** to work properly on such *reader* functions. Also, unless overridden, it defines a predicate named *name*`-p`, defines a constructor function named `make-`*constructor-name*, and defines a copier function named `copy-`*constructor-name*. All names of automatically created functions might automatically be declared **inline** (at the discretion of the *implementation*).

If *documentation* is supplied, it is attached to *structure-name* as a *documentation string* of kind **structure**, and unless `:type` is used, the *documentation* is also attached to *structure-name* as a *documentation string* of kind **type** and as a *documentation string* to the *class object* for the *class* named *structure-name*.

defstruct defines a constructor function that is used to create instances of the structure created by **defstruct**. The default name is make-*structure-name*. A different name can be supplied by giving the name as the argument to the *constructor* option. **nil** indicates that no constructor function will be created.

After a new structure type has been defined, instances of that type normally can be created by using the constructor function for the type. A call to a constructor function is of the following form:

```
(constructor-function-name
 slot-keyword-1 form-1
 slot-keyword-2 form-2
 ...)
```

The arguments to the constructor function are all keyword arguments. Each slot keyword argument must be a keyword whose name corresponds to the name of a structure slot. All the *keywords* and *forms* are evaluated. If a slot is not initialized in this way, it is initialized by evaluating *slot-initform* in the slot description at the time the constructor function is called. If no *slot-initform* is supplied, the consequences are undefined if an attempt is later made to read the slot's value before a value is explicitly assigned.

Each *slot-initform* supplied for a **defstruct** component, when used by the constructor function for an otherwise unsupplied component, is re-evaluated on every call to the constructor function. The *slot-initform* is not evaluated unless it is needed in the creation of a particular structure instance. If it is never needed, there can be no type-mismatch error, even if the *type* of the slot is specified; no warning should be issued in this case. For example, in the following sequence, only the last call is an error.

```
(defstruct person (name 007 :type string))
(make-person :name "James")
(make-person)
```

It is as if the *slot-initforms* were used as *initialization forms* for the *keyword parameters* of the constructor function.

The *symbols* which name the slots must not be used by the *implementation* as the *names* for the *lambda variables* in the constructor function, since one or more of those *symbols* might have been proclaimed **special** or might be defined as the name of a *constant variable*. The slot default init forms are evaluated in the *lexical environment* in which the **defstruct** form itself appears and in the *dynamic environment* in which the call to the constructor function appears.

For example, if the form (gensym) were used as an initialization form, either in the constructor-function call or as the default initialization form in **defstruct**, then every call to the constructor function would call **gensym** once to generate a new *symbol*.

defstruct

Each *slot-description* in **defstruct** can specify zero or more *slot-options*. A *slot-option* consists of a pair of a keyword and a value (which is not a form to be evaluated, but the value itself). For example:

```
(defstruct ship
  (x-position 0.0 :type short-float)
  (y-position 0.0 :type short-float)
  (x-velocity 0.0 :type short-float)
  (y-velocity 0.0 :type short-float)
  (mass *default-ship-mass* :type short-float :read-only t))
```

This specifies that each slot always contains a *short float*, and that the last slot cannot be altered once a ship is constructed.

The available slot-options are:

`:type` *type*

> This specifies that the contents of the slot is always of type *type*. This is entirely analogous to the declaration of a variable or function; it effectively declares the result type of the *reader* function. It is *implementation-dependent* whether the *type* is checked when initializing a slot or when assigning to it. *Type* is not evaluated; it must be a valid *type specifier*.

`:read-only` *x*

> When *x* is *true*, this specifies that this slot cannot be altered; it will always contain the value supplied at construction time. **setf** will not accept the *reader* function for this slot. If *x* is *false*, this slot-option has no effect. *X* is not evaluated.
>
> When this option is *false* or unsupplied, it is *implementation-dependent* whether the ability to *write* the slot is implemented by a *setf function* or a *setf expander*.

The following keyword options are available for use with **defstruct**. A **defstruct** option can be either a keyword or a *list* of a keyword and arguments for that keyword; specifying the keyword by itself is equivalent to specifying a list consisting of the keyword and no arguments. The syntax for **defstruct** options differs from the pair syntax used for slot-options. No part of any of these options is evaluated.

`:conc-name`

> This provides for automatic prefixing of names of *reader* (or *access*) functions. The default behavior is to begin the names of all the *reader* functions of a structure with the name of the structure followed by a hyphen.
>
> `:conc-name` supplies an alternate prefix to be used. If a hyphen is to be used as a separator, it must be supplied as part of the prefix. If `:conc-name` is **nil** or no argument is supplied, then no prefix is used; then the names of the *reader* functions are the same

as the slot names. If a *non-nil* prefix is given, the name of the *reader function* for each slot is constructed by concatenating that prefix and the name of the slot, and interning the resulting *symbol* in the *package* that is current at the time the **defstruct** form is expanded.

Note that no matter what is supplied for `:conc-name`, slot keywords that match the slot names with no prefix attached are used with a constructor function. The *reader* function name is used in conjunction with **setf**. Here is an example:

```
 (defstruct (door (:conc-name dr-)) knob-color width material) → DOOR
 (setq my-door (make-door :knob-color 'red :width 5.0))
→ #S(DOOR :KNOB-COLOR RED :WIDTH 5.0 :MATERIAL NIL)
 (dr-width my-door) → 5.0
 (setf (dr-width my-door) 43.7) → 43.7
 (dr-width my-door) → 43.7
```

Whether or not the `:conc-name` option is explicitly supplied, the following rule governs name conflicts of generated *reader* (or *accessor*) names: For any *structure type* S_1 having a *reader* function named R for a slot named X_1 that is inherited by another *structure type* S_2 that would have a *reader* function with the same name R for a slot named X_2, no definition for R is generated by the definition of S_2; instead, the definition of R is inherited from the definition of S_1. (In such a case, if X_1 and X_2 are different slots, the *implementation* might signal a style warning.)

`:constructor`

This option takes zero, one, or two arguments. If at least one argument is supplied and the first argument is not **nil**, then that argument is a *symbol* which specifies the name of the constructor function. If the argument is not supplied (or if the option itself is not supplied), the name of the constructor is produced by concatenating the string `"MAKE-"` and the name of the structure, interning the name in whatever *package* is current at the time **defstruct** is expanded. If the argument is provided and is **nil**, no constructor function is defined.

If `:constructor` is given as `(:constructor` *name arglist*`)`, then instead of making a keyword driven constructor function, **defstruct** defines a "positional" constructor function, taking arguments whose meaning is determined by the argument's position and possibly by keywords. *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 function whose arguments are used to initialize the slots named `a`, `b`, and `c`.

Because a constructor of this type operates "By Order of Arguments," it is sometimes known as a "boa constructor."

For information on how the *arglist* for a "boa constructor" is processed, see Section 3.4.6 (Boa Lambda Lists).

defstruct

It is permissible to use the `:constructor` option more than once, so that you can define several different constructor functions, each taking different parameters.

defstruct creates the default-named keyword constructor function only if no explicit `:constructor` options are specified, or if the `:constructor` option is specified without a *name* argument.

`(:constructor nil)` is meaningful only when there are no other `:constructor` options specified. It prevents **defstruct** from generating any constructors at all.

Otherwise, **defstruct** creates a constructor function corresponding to each supplied `:constructor` option. It is permissible to specify multiple keyword constructor functions as well as multiple "boa constructors".

`:copier`

This option takes one argument, a *symbol*, which specifies the name of the copier function. If the argument is not provided or if the option itself is not provided, the name of the copier is produced by concatenating the string `"COPY-"` and the name of the structure, interning the name in whatever *package* is current at the time **defstruct** is expanded. If the argument is provided and is **nil**, no copier function is defined.

The automatically defined copier function is a function of one *argument*, which must be of the structure type being defined. The copier function creates a *fresh* structure that has the same *type* as its *argument*, and that has the *same* component values as the original structure; that is, the component values are not copied recursively. If the **defstruct** `:type` option was not used, the following equivalence applies:

```
(copier-name x) = (copy-structure (the structure-name x))
```

`:include`

This option is used for building a new structure definition as an extension of another structure definition. For example:

```
(defstruct person name age sex)
```

To make a new structure to represent an astronaut that has the attributes of name, age, and sex, and *functions* that operate on `person` structures, `astronaut` is defined with `:include` as follows:

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

`:include` causes the structure being defined to have the same slots as the included structure. This is done in such a way that the *reader* functions for the included structure also work on the structure being defined. In this example, an `astronaut` therefore has five slots: the three defined in `person` and the two defined in `astronaut` itself. The *reader* functions defined by the `person` structure can be applied to instances of the `astronaut` structure, and they work correctly. Moreover, `astronaut` has its own *reader* functions for components defined by the `person` structure. The following examples illustrate the use of `astronaut` structures:

```
(setq x (make-astronaut :name 'buzz
                        :age 45.
                        :sex t
                        :helmet-size 17.5))
(person-name x) → BUZZ
(astro-name x) → BUZZ
(astro-favorite-beverage x) → TANG


(reduce #'+ astros :key #'person-age) ; obtains the total of the ages
                                      ; of the possibly empty
                                      ; sequence of astros
```

The difference between the *reader* functions `person-name` and `astro-name` is that `person-name` can be correctly applied to any `person`, including an `astronaut`, while `astro-name` can be correctly applied only to an `astronaut`. An implementation might check for incorrect use of *reader* functions.

At most one `:include` can be supplied in a single **defstruct**. The argument to `:include` is required and must be the name of some previously defined structure. If the structure being defined has no `:type` option, then the included structure must also have had no `:type` option supplied for it. If the structure being defined has a `:type` option, then the included structure must have been declared with a `:type` option specifying the same representation *type*.

If no `:type` option is involved, then the structure name of the including structure definition becomes the name of a *data type*, and therefore a valid *type specifier* recognizable by **typep**; it becomes a *subtype* of the included structure. In the above example, `astronaut` is a *subtype* of `person`; hence

`(typep (make-astronaut) 'person)` → *true*

indicating that all operations on persons also work on astronauts.

The structure using `:include` can specify default values or slot-options for the included slots different from those the included structure specifies, by giving the `:include` option as:

`(:include` *included-structure-name* {*slot-description*}*`)`

Each *slot-description* must have a *slot-name* that is the same as that of some slot in the included structure. If a *slot-description* has no *slot-initform*, then in the new structure the

defstruct

slot has no initial value. Otherwise its initial value form is replaced by the *slot-initform* in the *slot-description*. A normally writable slot can be made read-only. If a slot is read-only in the included structure, then it must also be so in the including structure. If a *type* is supplied for a slot, it must be a *subtype* of the *type* specified in the included structure.

For example, if the default age for an astronaut is `45`, then

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

If `:include` is used with the `:type` option, then the effect is first to skip over as many representation elements as needed to represent the included structure, then to skip over any additional elements supplied by the `:initial-offset` option, and then to begin allocation of elements from that point. For example:

```
(defstruct (binop (:type list) :named (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2) → BINOP
(defstruct (annotated-binop (:type list)
                            (:initial-offset 3)
                            (:include binop))
 commutative associative identity) → ANNOTATED-BINOP
(make-annotated-binop :operator '*
                      :operand-1 'x
                      :operand-2 5
                      :commutative t
                      :associative t
                      :identity 1)
  → (NIL NIL BINOP * X 5 NIL NIL NIL T T 1)
```

The first two **nil** elements stem from the `:initial-offset` of `2` in the definition of `binop`. The next four elements contain the structure name and three slots for `binop`. The next three **nil** elements stem from the `:initial-offset` of `3` in the definition of `annotated-binop`. The last three list elements contain the additional slots for an `annotated-binop`.

`:initial-offset`

`:initial-offset` instructs **defstruct** to skip over a certain number of slots before it starts allocating the slots described in the body. This option's argument is the number of slots **defstruct** should skip. `:initial-offset` can be used only if `:type` is also supplied.

`:initial-offset` allows slots to be allocated beginning at a representational element other than the first. For example, the form

```
(defstruct (binop (:type list) (:initial-offset 2))
```

```
  (operator '? :type symbol)
  operand-1
  operand-2) → BINOP
```

would result in the following behavior for `make-binop`:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5)
→ (NIL NIL + X 5)
(make-binop :operand-2 4 :operator '*)
→ (NIL NIL * NIL 4)
```

The selector functions `binop-operator`, `binop-operand-1`, and `binop-operand-2` would be essentially equivalent to **third**, **fourth**, and **fifth**, respectively. Similarly, the form

```
(defstruct (binop (:type list) :named (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2) → BINOP
```

would result in the following behavior for `make-binop`:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5) → (NIL NIL BINOP + X 5)
(make-binop :operand-2 4 :operator '*) → (NIL NIL BINOP * NIL 4)
```

The first two **nil** elements stem from the `:initial-offset` of `2` in the definition of `binop`. The next four elements contain the structure name and three slots for `binop`.

`:named`

`:named` specifies that the structure is named. If no `:type` is supplied, then the structure is always named.

For example:

```
(defstruct (binop (:type list))
  (operator '? :type symbol)
  operand-1
  operand-2) → BINOP
```

This defines a constructor function `make-binop` and three selector functions, namely `binop-operator`, `binop-operand-1`, and `binop-operand-2`. (It does not, however, define a predicate `binop-p`, for reasons explained below.)

The effect of `make-binop` is simply to construct a list of length three:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5) → (+ X 5)
(make-binop :operand-2 4 :operator '*) → (* NIL 4)
```

It is just like the function `list` except that it takes keyword arguments and performs slot defaulting appropriate to the `binop` conceptual data type. Similarly, the selector functions

defstruct

`binop-operator`, `binop-operand-1`, and `binop-operand-2` are essentially equivalent to **car**, **cadr**, and **caddr**, respectively. They might not be completely equivalent because, for example, an implementation would be justified in adding error-checking code to ensure that the argument to each selector function is a length-3 list.

`binop` is a conceptual data type in that it is not made a part of the Common Lisp type system. **typep** does not recognize `binop` as a *type specifier*, and **type-of** returns `list` when given a `binop` structure. There is no way to distinguish a data structure constructed by `make-binop` from any other *list* that happens to have the correct structure.

There is not any way to recover the structure name `binop` from a structure created by `make-binop`. This can only be done if the structure is named. A named structure has the property that, given an instance of the structure, the structure name (that names the type) can be reliably recovered. For structures defined with no `:type` option, the structure name actually becomes part of the Common Lisp data-type system. **type-of**, when applied to such a structure, returns the structure name as the *type* of the *object*; **typep** recognizes the structure name as a valid *type specifier*.

For structures defined with a `:type` option, **type-of** returns a *type specifier* such as `list` or `(vector t)`, depending on the type supplied to the `:type` option. The structure name does not become a valid *type specifier*. However, if the `:named` option is also supplied, then the first component of the structure (as created by a **defstruct** constructor function) always contains the structure name. This allows the structure name to be recovered from an instance of the structure and allows a reasonable predicate for the conceptual type to be defined: the automatically defined *name-p* predicate for the structure operates by first checking that its argument is of the proper type (**list**, `(vector t)`, or whatever) and then checking whether the first component contains the appropriate type name.

Consider the `binop` example shown above, modified only to include the `:named` option:

```
(defstruct (binop (:type list) :named)
  (operator '? :type symbol)
  operand-1
  operand-2) → BINOP
```

As before, this defines a constructor function `make-binop` and three selector functions `binop-operator`, `binop-operand-1`, and `binop-operand-2`. It also defines a predicate `binop-p`. The effect of `make-binop` is now to construct a list of length four:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5) → (BINOP + X 5)
(make-binop :operand-2 4 :operator '*) → (BINOP * NIL 4)
```

The structure has the same layout as before except that the structure name `binop` is included as the first list element. The selector functions `binop-operator`, `binop-operand-1`, and `binop-operand-2` are essentially equivalent to **cadr**, **caddr**, and **cadddr**, respectively. The predicate `binop-p` is more or less equivalent to this definition:

```
(defun binop-p (x)
```

defstruct

```
(and (consp x) (eq (car x) 'binop))) → BINOP-P
```

The name `binop` is still not a valid *type specifier* recognizable to **typep**, but at least there is a way of distinguishing `binop` structures from other similarly defined structures.

`:predicate`

This option takes one argument, which specifies the name of the type predicate. If the argument is not supplied or if the option itself is not supplied, the name of the predicate is made by concatenating the name of the structure to the string `"-P"`, interning the name in whatever *package* is current at the time **defstruct** is expanded. If the argument is provided and is **nil**, no predicate is defined. A predicate can be defined only if the structure is named; if `:type` is supplied and `:named` is not supplied, then `:predicate` must either be unsupplied or have the value **nil**.

`:print-function`, `:print-object`

The `:print-function` and `:print-object` options specify that a **print-object** *method* for *structures* of type *structure-name* should be generated. These options are not synonyms, but do perform a similar service; the choice of which option (`:print-function` or `:print-object`) is used affects how the function named *printer-name* is called. Only one of these options may be used, and these options may be used only if `:type` is not supplied.

If the `:print-function` option is used, then when a structure of type *structure-name* is to be printed, the designated printer function is called on three *arguments*:

– the structure to be printed (a *generalized instance* of *structure-name*).

– a *stream* to print to.

– an *integer* indicating the current depth. The magnitude of this integer may vary between *implementations*; however, it can reliably be compared against ***print-level*** to determine whether depth abbreviation is appropriate.

Specifying `(:print-function` *printer-name*`)` is approximately equivalent to specifying:

```
(defmethod print-object ((object structure-name) stream)
  (funcall (function printer-name) object stream ⟨⟨current-print-depth⟩⟩))
```

where the ⟨⟨*current-print-depth*⟩⟩ represents the printer's belief of how deep it is currently printing. It is *implementation-dependent* whether ⟨⟨*current-print-depth*⟩⟩ is always 0 and **print-level**, if *non-nil*, is re-bound to successively smaller values as printing descends recursively, or whether *current-print-depth* varies in value as printing descends recursively and **print-level** remains constant during the same traversal.

If the `:print-object` option is used, then when a structure of type *structure-name* is to be printed, the designated printer function is called on two arguments:

defstruct

– the structure to be printed.

– the stream to print to.

Specifying (`:print-object` *printer-name*) is equivalent to specifying:

```
(defmethod print-object ((object structure-name) stream)
  (funcall (function printer-name) object stream))
```

If no `:type` option is supplied, and if either a `:print-function` or a `:print-object` option is supplied, and if no *printer-name* is supplied, then a **print-object** *method specialized* for *structure-name* is generated that calls a function that implements the default printing behavior for structures using `#S` notation; see Section 22.1.3.12 (Printing Structures).

If neither a `:print-function` nor a `:print-object` option is supplied, then **defstruct** does not generate a **print-object** *method specialized* for *structure-name* and some default behavior is inherited either from a structure named in an `:include` option or from the default behavior for printing structures; see the *function* **print-object** and Section 22.1.3.12 (Printing Structures).

When ***print-circle*** is *true*, a user-defined print function can print *objects* to the supplied *stream* using **write**, **prin1**, **princ**, or **format** and expect circularities to be detected and printed using the `#n#` syntax. This applies to *methods* on **print-object** in addition to `:print-function` options. If a user-defined print function prints to a *stream* other than the one that was supplied, then circularity detection starts over for that *stream*. See the *variable* ***print-circle***.

`:type`

`:type` explicitly specifies the representation to be used for the structure. Its argument must be one of these *types*:

vector

This produces the same result as specifying `(vector t)`. The structure is represented as a general *vector*, storing components as vector elements. The first component is vector element 1 if the structure is `:named`, and element 0 otherwise.

`(vector` *element-type*`)`

The structure is represented as a (possibly specialized) *vector*, storing components as vector elements. Every component must be of a *type* that can be stored in a *vector* of the *type* specified. The first component is vector element 1 if the structure is `:named`, and element 0 otherwise. The structure can be `:named` only if the *type* **symbol** is a *subtype* of the supplied *element-type*.

list

defstruct

The structure is represented as a *list*. The first component is the *cadr* if the structure is `:named`, and the *car* if it is not `:named`.

Specifying this option has the effect of forcing a specific representation and of forcing the components to be stored in the order specified in **defstruct** in corresponding successive elements of the specified representation. It also prevents the structure name from becoming a valid *type specifier* recognizable by **typep**.

For example:

```
(defstruct (quux (:type list) :named) x y)
```

should make a constructor that builds a *list* exactly like the one that **list** produces, with `quux` as its *car*.

If this type is defined:

```
(deftype quux () '(satisfies quux-p))
```

then this form

```
(typep (make-quux) 'quux)
```

should return precisely what this one does

```
(typep (list 'quux nil nil) 'quux)
```

If `:type` is not supplied, the structure is represented as an *object* of *type* **structure-object**.

defstruct without a `:type` option defines a *class* with the structure name as its name. The *metaclass* of structure *instances* is **structure-class**.

The consequences of redefining a **defstruct** structure are undefined.

In the case where no **defstruct** options have been supplied, the following functions are automatically defined to operate on instances of the new structure:

Predicate

A predicate with the name *structure-name*-`p` is defined to test membership in the structure type. The predicate `(`*structure-name*-`p` *object*`)` is *true* if an *object* is of this *type*; otherwise it is *false*. **typep** can also be used with the name of the new *type* to test whether an *object* belongs to the *type*. Such a function call has the form `(typep` *object* `'`*structure-name*`)`.

Component reader functions

Reader functions are defined to *read* the components of the structure. For each slot name, there is a corresponding *reader* function with the name *structure-name-slot-name*.

defstruct

This function *reads* the contents of that slot. Each *reader* function takes one argument, which is an instance of the structure type. **setf** can be used with any of these *reader* functions to alter the slot contents.

Constructor function

A constructor function with the name `make-`*structure-name* is defined. This function creates and returns new instances of the structure type.

Copier function

A copier function with the name `copy-`*structure-name* is defined. The copier function takes an object of the structure type and creates a new object of the same type that is a copy of the first. The copier function creates a new structure with the same component entries as the original. Corresponding components of the two structure instances are **eql**.

If a **defstruct** *form* appears as a *top level form*, the *compiler* must make the *structure type* name recognized as a valid *type* name in subsequent declarations (as for **deftype**) and make the structure slot readers known to **setf**. In addition, the *compiler* must save enough information about the *structure type* so that further **defstruct** definitions can use `:include` in a subsequent **deftype** in the same *file* to refer to the *structure type* name. The functions which **defstruct** generates are not defined in the compile time environment, although the *compiler* may save enough information about the functions to code subsequent calls inline. The `#S` *reader macro* might or might not recognize the newly defined *structure type* name at compile time.

Examples:

An example of a structure definition follows:

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

This declares that every `ship` is an *object* with five named components. The evaluation of this form does the following:

1. It defines `ship-x-position` to be a function of one argument, a ship, that returns the `x-position` of the ship; `ship-y-position` and the other components are given similar function definitions. These functions are called the *access* functions, as they are used to *access* elements of the structure.

2. `ship` becomes the name of a *type* of which instances of ships are elements. `ship` becomes acceptable to **typep**, for example; `(typep x 'ship)` is *true* if `x` is a ship and false if `x` is any *object* other than a ship.

3. A function named `ship-p` of one argument is defined; it is a predicate that is *true* if its argument is a ship and is *false* otherwise.

4. A function called `make-ship` is defined that, when invoked, creates a data structure with five components, suitable for use with the *access* functions. Thus executing

   ```
   (setq ship2 (make-ship))
   ```

   sets `ship2` to a newly created `ship` *object*. One can supply the initial values of any desired component in the call to `make-ship` by using keyword arguments in this way:

   ```
   (setq ship2 (make-ship :mass *default-ship-mass*
                          :x-position 0
                          :y-position 0))
   ```

   This constructs a new ship and initializes three of its components. This function is called the "constructor function" because it constructs a new structure.

5. A function called `copy-ship` of one argument is defined that, when given a `ship` *object*, creates a new `ship` *object* that is a copy of the given one. This function is called the "copier function."

setf can be used to alter the components of a `ship`:

```
(setf (ship-x-position ship2) 100)
```

This alters the `x-position` of `ship2` to be `100`. This works because **defstruct** behaves as if it generates an appropriate **defsetf** for each *access* function.

```
;;;
;;; Example 1
;;; define town structure type
;;; area, watertowers, firetrucks, population, elevation are its components
;;;
 (defstruct town
             area
             watertowers
             (firetrucks 1 :type fixnum)    ;an initialized slot
             population
             (elevation 5128 :read-only t)) ;a slot that can't be changed
→ TOWN
;create a town instance
 (setq town1 (make-town :area 0 :watertowers 0)) → #S(TOWN...)
;town's predicate recognizes the new instance
 (town-p town1) → true
;new town's area is as specified by make-town
 (town-area town1) → 0
;new town's elevation has initial value
```

defstruct

```
 (town-elevation town1) → 5128
;setf recognizes reader function
 (setf (town-population town1) 99) → 99
 (town-population town1) → 99
;copier function makes a copy of town1
 (setq town2 (copy-town town1)) → #S(TOWN...)
 (= (town-population town1) (town-population town2))  → true
;since elevation is a read-only slot, its value can be set only
;when the structure is created
 (setq town3 (make-town :area 0 :watertowers 3 :elevation 1200))
→ #S(TOWN...)
;;;
;;; Example 2
;;; define clown structure type
;;; this structure uses a nonstandard prefix
;;;
 (defstruct (clown (:conc-name bozo-))
            (nose-color 'red)
            frizzy-hair-p polkadots) → CLOWN
 (setq funny-clown (make-clown)) → #S(CLOWN)
;use non-default reader name
 (bozo-nose-color funny-clown) → RED
 (defstruct (klown (:constructor make-up-klown) ;similar def using other
            (:copier clone-klown)              ;customizing keywords
            (:predicate is-a-bozo-p))
            nose-color frizzy-hair-p polkadots) → klown
;custom constructor now exists
 (fboundp 'make-up-klown) → true
;;;
;;; Example 3
;;; define a vehicle structure type
;;; then define a truck structure type that includes
;;; the vehicle structure
;;;
 (defstruct vehicle name year (diesel t :read-only t)) → VEHICLE
 (defstruct (truck (:include vehicle (year 79)))
            load-limit
            (axles 6)) → TRUCK
 (setq x (make-truck :name 'mac :diesel t :load-limit 17))
→ #S(TRUCK...)
;vehicle readers work on trucks
 (vehicle-name x)
→ MAC
;default taken from :include clause
```

```
 (vehicle-year x)
→ 79
 (defstruct (pickup (:include truck))     ;pickup type includes truck
            camper long-bed four-wheel-drive) → PICKUP
 (setq x (make-pickup :name 'king :long-bed t)) → #S(PICKUP...)
;:include default inherited
 (pickup-year x) → 79
;;;
;;; Example 4
;;; use of BOA constructors
;;;
 (defstruct (dfs-boa                      ;BOA constructors
              (:constructor make-dfs-boa (a b c))
              (:constructor create-dfs-boa
                (a &optional b (c 'cc) &rest d &aux e (f 'ff))))
            a b c d e f) → DFS-BOA
;a, b, and c set by position, and the rest are uninitialized
 (setq x (make-dfs-boa 1 2 3)) → #(DFS-BOA...)
 (dfs-boa-a x) → 1
;a and b set, c and f defaulted
 (setq x (create-dfs-boa 1 2)) → #(DFS-BOA...)
 (dfs-boa-b x) → 2
 (eq (dfs-boa-c x) 'cc) → true
;a, b, and c set, and the rest are collected into d
 (setq x (create-dfs-boa 1 2 3 4 5 6)) → #(DFS-BOA...)
 (dfs-boa-d x) → (4 5 6)
```

Exceptional Situations:

If any two slot names (whether present directly or inherited by the `:include` option) are the *same* under **string=**, **defstruct** should signal an error of *type* **program-error**.

The consequences are undefined if the *included-structure-name* does not name a *structure type*.

See Also:

documentation, **print-object**, **setf**, **subtypep**, **type-of**, **typep**, Section 3.2 (Compilation)

Notes:

The *printer-name* should observe the values of such printer-control variables as ***print-escape***.

The restriction against issuing a warning for type mismatches between a *slot-initform* and the corresponding slot's `:type` option is necessary because a *slot-initform* must be specified in order to specify slot options; in some cases, no suitable default may exist.

The mechanism by which **defstruct** arranges for slot accessors to be usable with **setf** is *implementation-dependent*; for example, it may use *setf functions*, *setf expanders*, or some other *implementation-dependent* mechanism known to that *implementation*'s *code* for **setf**.

copy-structure *Function*

Syntax:

copy-structure *structure* → *copy*

Arguments and Values:

structure—a *structure*.

copy—a copy of the *structure*.

Description:

Returns a $copy_6$ of the *structure*.

Only the *structure* itself is copied; not the values of the slots.

See Also:

the `:copier` option to **defstruct**

Notes:

The *copy* is the *same* as the given *structure* under **equalp**, but not under **equal**.