3. Evaluation and Compilation

3.1 Evaluation

Execution of *code* can be accomplished by a variety of means ranging from direct interpretation of a *form* representing a *program* to invocation of *compiled code* produced by a *compiler*.

Evaluation is the process by which a *program* is *executed* in Common Lisp. The mechanism of *evaluation* is manifested both implicitly through the effect of the *Lisp read-eval-print loop*, and explicitly through the presence of the *functions* **eval**, **compile**, **compile-file**, and **load**. Any of these facilities might share the same execution strategy, or each might use a different one.

The behavior of a *conforming program* processed by **eval** and by **compile-file** might differ; see Section 3.2.2.3 (Semantic Constraints).

Evaluation can be understood in terms of a model in which an interpreter recursively traverses a *form* performing each step of the computation as it goes. This model, which describes the semantics of Common Lisp *programs*, is described in Section 3.1.2 (The Evaluation Model).

3.1.1 Introduction to Environments

A *binding* is an association between a *name* and that which the name denotes. *Bindings* are *established* in a *lexical environment* or a *dynamic environment* by particular *special operators*.

An *environment* is a set of *bindings* and other information used during evaluation (e.g., to associate meanings with names).

Bindings in an *environment* are partitioned into *namespaces*. A single *name* can simultaneously have more than one associated *binding* per *environment*, but can have only one associated *binding* per *namespace*.

3.1.1.1 The Global Environment

The *global environment* is that part of an *environment* that contains *bindings* with both *indefinite scope* and *indefinite extent*. The *global environment* contains, among other things, the following:

- *bindings* of *dynamic variables* and *constant variables*.
- *bindings* of *functions*, *macros*, and *special operators*.
- *bindings* of *compiler macros*.
- *bindings* of *type* and *class names*
- information about *proclamations*.

3.1.1.2 Dynamic Environments

A *dynamic environment* for *evaluation* is that part of an *environment* that contains *bindings* whose duration is bounded by points of *establishment* and *disestablishment* within the execution of the *form* that established the *binding*. A *dynamic environment* contains, among other things, the following:

- *bindings* for *dynamic variables*.
- information about *active catch tags*.
- information about *exit points* established by **unwind-protect**.
- information about *active handlers* and *restarts*.

The *dynamic environment* that is active at any given point in the *execution* of a *program* is referred to by definite reference as "the current *dynamic environment*," or sometimes as just "the *dynamic environment*."

Within a given *namespace*, a *name* is said to be *bound* in a *dynamic environment* if there is a *binding* associated with its *name* in the *dynamic environment* or, if not, there is a *binding* associated with its name in the *global environment*.

3.1.1.3 Lexical Environments

A *lexical environment* for *evaluation* at some position in a *program* is that part of the *environment* that contains information having *lexical scope* within the *forms* containing that position. A *lexical environment* contains, among other things, the following:

- *bindings* of *lexical variables* and *symbol macros*.
- *bindings* of *functions* and *macros*. (Implicit in this is information about those *compiler macros* that are locally disabled.)
- *bindings* of *block tags*.
- *bindings* of *go tags*.
- information about *declarations*.

The *lexical environment* that is active at any given position in a *program* being semantically processed is referred to by definite reference as "the current *lexical environment*," or sometimes as just "the *lexical environment*."

Within a given *namespace*, a *name* is said to be *bound* in a *lexical environment* if there is a *binding* associated with its *name* in the *lexical environment* or, if not, there is a *binding* associated with its name in the *global environment*.

3.1.1.3.1 The Null Lexical Environment

The *null lexical environment* is equivalent to the *global environment*.

Although in general the representation of an *environment object* is *implementation-dependent*, **nil** can be used in any situation where an *environment object* is called for in order to denote the *null lexical environment*.

3.1.1.4 Environment Objects

Some *operators* make use of an *object*, called an *environment object*, that represents the set of *lexical bindings* needed to perform semantic analysis on a *form* in a given *lexical environment*. The set of *bindings* in an *environment object* may be a subset of the *bindings* that would be needed to actually perform an *evaluation*; for example, *values* associated with *variable names* and *function names* in the corresponding *lexical environment* might not be available in an *environment object*.

The *type* and nature of an *environment object* is *implementation-dependent*. The *values* of *environment parameters* to *macro functions* are examples of *environment objects*.

The *object* **nil** when used as an *environment object* denotes the *null lexical environment*; see Section 3.1.1.3.1 (The Null Lexical Environment).

3.1.2 The Evaluation Model

A Common Lisp system evaluates *forms* with respect to lexical, dynamic, and global *environments*. The following sections describe the components of the Common Lisp evaluation model.

3.1.2.1 Form Evaluation

Forms fall into three categories: *symbols*, *conses*, and *self-evaluating objects*. The following sections explain these categories.

3.1.2.1.1 Symbols as Forms

If a *form* is a *symbol*, then it is either a *symbol macro* or a *variable*.

The *symbol* names a *symbol macro* if there is a *binding* of the *symbol* as a *symbol macro* in the current *lexical environment* (see **define-symbol-macro** and **symbol-macrolet**). If the *symbol* is a *symbol macro*, its expansion function is obtained. The expansion function is a function of two arguments, and is invoked by calling the *macroexpand hook* with the expansion function as its first argument, the *symbol* as its second argument, and an *environment object* (corresponding to the current *lexical environment*) as its third argument. The *macroexpand hook*, in turn, calls the expansion function with the *form* as its first argument and the *environment* as its second argument. The *value* of the expansion function, which is passed through by the *macroexpand hook*, is a *form*. This resulting *form* is processed in place of the original *symbol*.

If a *form* is a *symbol* that is not a *symbol macro*, then it is the *name* of a *variable*, and the *value* of that *variable* is returned. There are three kinds of variables: *lexical variables*, *dynamic variables*, and *constant variables*. A *variable* can store one *object*. The main operations on a *variable* are to *read*[1] and to *write*[1] its *value*.

An error of *type* **unbound-variable** should be signaled if an *unbound variable* is referenced.

Non-constant variables can be *assigned* by using **setq** or *bound*[3] by using **let**. The next figure lists some *defined names* that are applicable to assigning, binding, and defining *variables*.

```
boundp        let                 progv
defconstant   let*                psetq
defparameter  makunbound          set
defvar        multiple-value-bind setq
lambda        multiple-value-setq symbol-value
```

Figure 3-1. Some Defined Names Applicable to Variables

The following is a description of each kind of variable.

3.1.2.1.1.1 Lexical Variables

A *lexical variable* is a *variable* that can be referenced only within the *lexical scope* of the *form* that establishes that *variable*; *lexical variables* have *lexical scope*. Each time a *form* creates a *lexical binding* of a *variable*, a *fresh binding* is *established*.

Within the *scope* of a *binding* for a *lexical variable name*, uses of that *name* as a *variable* are considered to be references to that *binding* except where the *variable* is *shadowed*[2] by a *form* that *establishes* a *fresh binding* for that *variable name*, or by a *form* that locally *declares* the *name* **special**.

A *lexical variable* always has a *value*. There is no *operator* that introduces a *binding* for a *lexical variable* without giving it an initial *value*, nor is there any *operator* that can make a *lexical variable* be *unbound*.

Bindings of *lexical variables* are found in the *lexical environment*.

3.1.2.1.1.2 Dynamic Variables

A *variable* is a *dynamic variable* if one of the following conditions hold:

- It is locally declared or globally proclaimed **special**.
- It occurs textually within a *form* that creates a *dynamic binding* for a *variable* of the *same name*, and the *binding* is not *shadowed*[2] by a *form* that creates a *lexical binding* of the same *variable name*.

A *dynamic variable* can be referenced at any time in any *program*; there is no textual limitation on references to *dynamic variables*. At any given time, all *dynamic variables* with a given name refer to exactly one *binding*, either in the *dynamic environment* or in the *global environment*.

The *value* part of the *binding* for a *dynamic variable* might be empty; in this case, the *dynamic variable* is said to have no *value*, or to be *unbound*. A *dynamic variable* can be made *unbound* by using **makunbound**.

The effect of *binding* a *dynamic variable* is to create a new *binding* to which all references to that *dynamic variable* in any *program* refer for the duration of the *evaluation* of the *form* that creates the *dynamic binding*.

A *dynamic variable* can be referenced outside the *dynamic extent* of a *form* that *binds* it. Such a *variable* is sometimes called a "global variable" but is still in all respects just a *dynamic variable* whose *binding* happens to exist in the *global environment* rather than in some *dynamic environment*.

A *dynamic variable* is *unbound* unless and until explicitly assigned a value, except for those variables whose initial value is defined in this specification or by an *implementation*.

3.1.2.1.1.3 Constant Variables

Certain variables, called *constant variables*, are reserved as "named constants." The consequences are undefined if an attempt is made to assign a value to, or create a *binding* for a *constant variable*, except that a ‘compatible’ redefinition of a *constant variable* using **defconstant** is permitted; see the *macro* **defconstant**.

Keywords, *symbols* defined by Common Lisp or the *implementation* as constant (such as **nil**, **t**, and **pi**), and *symbols* declared as constant using **defconstant** are *constant variables*.

3.1.2.1.1.4 Symbols Naming Both Lexical and Dynamic Variables

The same *symbol* can name both a *lexical variable* and a *dynamic variable*, but never in the same *lexical environment*.

In the following example, the *symbol* `X` is used, at different times, as the *name* of a *lexical variable* and as the *name* of a *dynamic variable*.

```
 (let ((x 1))            ;Binds a special variable X
   (declare (special x))
   (let ((x 2))          ;Binds a lexical variable X
     (+ x                ;Reads a lexical variable X
        (locally (declare (special x))
                 x))))   ;Reads a special variable X
=>  3
```

3.1.2.1.2 Conses as Forms

A *cons* that is used as a *form* is called a *compound form*.

If the *car* of that *compound form* is a *symbol*, that *symbol* is the *name* of an *operator*, and the *form* is either a *special form*, a *macro form*, or a *function form*, depending on the *function binding* of the *operator* in the current *lexical environment*. If the *operator* is neither a *special operator* nor a *macro name*, it is assumed to be a *function name* (even if there is no definition for such a *function*).

If the *car* of the *compound form* is not a *symbol*, then that *car* must be a *lambda expression*, in which case the *compound form* is a *lambda form*.

How a *compound form* is processed depends on whether it is classified as a *special form*, a *macro form*, a *function form*, or a *lambda form*.

3.1.2.1.2.1 Special Forms

A *special form* is a *form* with special syntax, special evaluation rules, or both, possibly manipulating the evaluation environment, control flow, or both. A *special operator* has access to the current *lexical environment* and the current *dynamic environment*. Each *special operator* defines the manner in which its *subexpressions* are treated---which are *forms*, which are special syntax, *etc*.

Some *special operators* create new lexical or dynamic *environments* for use during the *evaluation* of *subforms* of the *special form*. For example, **block** creates a new *lexical environment* that is the same as the one in force at the point of evaluation of the **block** *form* with the addition of a *binding* of the **block** name to an *exit point* from the **block**.

The set of *special operator names* is fixed in Common Lisp; no way is provided for the user to define a *special operator*. The next figure lists all of the Common Lisp *symbols* that have definitions as *special operators*.

```
block      let*                  return-from
catch      load-time-value       setq
eval-when  locally               symbol-macrolet
flet       macrolet              tagbody
function   multiple-value-call   the
go         multiple-value-prog1  throw
if         progn                 unwind-protect
labels     progv
let        quote
```

Figure 3-2. Common Lisp Special Operators

3.1.2.1.2.2 Macro Forms

If the *operator* names a *macro*, its associated *macro function* is applied to the entire *form* and the result of that application is used in place of the original *form*.

Specifically, a *symbol* names a *macro* in a given *lexical environment* if **macro-function** is *true* of the *symbol* and that *environment*. The *function* returned by **macro-function** is a *function* of two arguments, called the expansion function. The expansion function is invoked by calling the *macroexpand hook* with the expansion function as its first argument, the entire *macro form* as its second argument, and an *environment object* (corresponding to the current *lexical environment*) as its third argument. The *macroexpand hook*, in turn, calls the expansion function with the *form* as its first argument and the *environment* as its second argument. The *value* of the expansion function, which is passed through by the *macroexpand hook*, is a *form*. The returned *form* is *evaluated* in place of the original *form*.

The consequences are undefined if a *macro function* destructively modifies any part of its *form* argument.

A *macro name* is not a *function designator*, and cannot be used as the *function* argument to *functions* such as **apply**, **funcall**, or **map**.

An *implementation* is free to implement a Common Lisp *special operator* as a *macro*. An *implementation* is free to implement any *macro operator* as a *special operator*, but only if an equivalent definition of the *macro* is also provided.

The next figure lists some *defined names* that are applicable to *macros*.

```
*macroexpand-hook*  macro-function  macroexpand-1
defmacro            macroexpand     macrolet
```

Figure 3-3. Defined names applicable to macros

3.1.2.1.2.3 Function Forms

If the *operator* is a *symbol* naming a *function*, the *form* represents a *function form*, and the *cdr* of the list contains the *forms* which when evaluated will supply the arguments passed to the *function*.

When a *function name* is not defined, an error of *type* **undefined-function** should be signaled at run time; see Section 3.2.2.3 (Semantic Constraints).

A *function form* is evaluated as follows:

The *subforms* in the *cdr* of the original *form* are evaluated in left-to-right order in the current lexical and dynamic *environments*. The *primary value* of each such *evaluation* becomes an *argument* to the named *function*; any additional *values* returned by the *subforms* are discarded.

The *functional value* of the *operator* is retrieved from the *lexical environment*, and that *function* is invoked with the indicated arguments.

Although the order of *evaluation* of the *argument subforms* themselves is strictly left-to-right, it is not specified whether the definition of the *operator* in a *function form* is looked up before the *evaluation* of the *argument subforms*, after the *evaluation* of the *argument subforms*, or between the *evaluation* of any two *argument subforms* if there is more than one such *argument subform*. For example, the following might return 23 or 24.

```
(defun foo (x) (+ x 3))
(defun bar () (setf (symbol-function 'foo) #'(lambda (x) (+ x 4))))
(foo (progn (bar) 20))
```

A *binding* for a *function name* can be *established* in one of several ways. A *binding* for a *function name* in the *global environment* can be *established* by **defun**, **setf** of **fdefinition**, **setf** of **symbol-function**, **ensure-generic-function**, **defmethod** (implicitly, due to **ensure-generic-function**), or **defgeneric**. A *binding* for a *function name* in the *lexical environment* can be *established* by **flet** or **labels**.

The next figure lists some *defined names* that are applicable to *functions*.

```
apply                 fdefinition  mapcan
call-arguments-limit  flet         mapcar
complement            fmakunbound  mapcon
constantly            funcall      mapl
defgeneric            function     maplist
defmethod             functionp    multiple-value-call
defun                 labels       reduce
fboundp               map          symbol-function
```

Figure 3-4. Some function-related defined names

3.1.2.1.2.4 Lambda Forms

A *lambda form* is similar to a *function form*, except that the *function name* is replaced by a *lambda expression*.

A *lambda form* is equivalent to using *funcall* of a *lexical closure* of the *lambda expression* on the given *arguments*. (In practice, some compilers are more likely to produce inline code for a *lambda form* than for an arbitrary named function that has been declared **inline**; however, such a difference is not semantic.)

For further information, see Section 3.1.3 (Lambda Expressions).

3.1.2.1.3 Self-Evaluating Objects

A *form* that is neither a *symbol* nor a *cons* is defined to be a *self-evaluating object*. *Evaluating* such an *object yields* the *same object* as a result.

Certain specific *symbols* and *conses* might also happen to be "self-evaluating" but only as a special case of a more general set of rules for the *evaluation* of *symbols* and *conses*; such *objects* are not considered to be *self-evaluating objects*.

The consequences are undefined if *literal objects* (including *self-evaluating objects*) are destructively modified.

3.1.2.1.3.1 Examples of Self-Evaluating Objects

Numbers, *pathnames*, and *arrays* are examples of *self-evaluating objects*.

```
3 =>  3
#c(2/3 5/8) =>  #C(2/3 5/8)
#p"S:[BILL]OTHELLO.TXT" =>  #P"S:[BILL]OTHELLO.TXT"
#(a b c) =>  #(A B C)
"fred smith" =>  "fred smith"
```

3.1.3 Lambda Expressions

In a *lambda expression*, the body is evaluated in a lexical *environment* that is formed by adding the *binding* of each *parameter* in the *lambda list* with the corresponding *value* from the *arguments* to the current lexical *environment*.

For further discussion of how *bindings* are *established* based on the *lambda list*, see Section 3.4 (Lambda Lists).

The body of a *lambda expression* is an *implicit progn*; the *values* it returns are returned by the *lambda expression*.

3.1.4 Closures and Lexical Binding

A *lexical closure* is a *function* that can refer to and alter the values of *lexical bindings established* by *binding forms* that textually include the function definition.

Consider this code, where `x` is not declared **special**:

```
(defun two-funs (x)
  (list (function (lambda () x))
        (function (lambda (y) (setq x y)))))
(setq funs (two-funs 6))
(funcall (car funs)) =>  6
(funcall (cadr funs) 43) =>  43
(funcall (car funs)) =>  43
```

The **function** *special form* coerces a *lambda expression* into a *closure* in which the *lexical environment* in effect when the *special form* is evaluated is captured along with the *lambda expression*.

The function `two-funs` returns a *list* of two *functions*, each of which refers to the *binding* of the variable `x` created on entry to the function `two-funs` when it was called. This variable has the value `6` initially, but **setq** can alter this *binding*. The *lexical closure* created for the first *lambda expression* does not "snapshot" the *value* `6` for `x` when the *closure* is created; rather it captures the *binding* of `x`. The second *function* can be used to alter the *value* in the same (captured) *binding* (to `43`, in the example), and this altered variable binding then affects the value returned by the first *function*.

In situations where a *closure* of a *lambda expression* over the same set of *bindings* may be produced more than once, the various resulting *closures* may or may not be *identical*, at the discretion of the *implementation*. That is, two *functions* that are behaviorally indistinguishable might or might not be *identical*. Two *functions* that are behaviorally distinguishable are *distinct*. For example:

```
(let ((x 5) (funs '()))
  (dotimes (j 10)
    (push #'(lambda (z)
              (if (null z) (setq x 0) (+ x z)))
          funs))
  funs)
```

The result of the above *form* is a *list* of ten *closures*. Each requires only the *binding* of `x`. It is the same *binding* in each case, but the ten *closure objects* might or might not be *identical*. On the other hand, the result of the *form*

```
(let ((funs '()))
  (dotimes (j 10)
    (let ((x 5))
      (push (function (lambda (z)
                        (if (null z) (setq x 0) (+ x z))))
            funs)))
 funs)
```

is also a *list* of ten *closures*. However, in this case no two of the *closure objects* can be *identical* because each *closure* is closed over a distinct *binding* of `x`, and these *bindings* can be behaviorally distinguished because of the use of **setq**.

The result of the *form*

```
(let ((funs '()))
  (dotimes (j 10)
    (let ((x 5))
      (push (function (lambda (z) (+ x z)))
           funs)))
  funs)
```

is a *list* of ten *closure objects* that might or might not be *identical*. A different *binding* of `x` is involved for each *closure*, but the *bindings* cannot be distinguished because their values are the *same* and immutable (there being no occurrence of **setq** on `x`). A compiler could internally transform the *form* to

```
(let ((funs '()))
  (dotimes (j 10)
    (push (function (lambda (z) (+ 5 z)))
          funs))
 funs)
```

where the *closures* may be *identical*.

It is possible that a *closure* does not close over any variable bindings. In the code fragment

```
(mapcar (function (lambda (x) (+ x 2))) y)
```

the function `(lambda (x) (+ x 2))` contains no references to any outside object. In this case, the same *closure* might be returned for all evaluations of the **function** *form*.

3.1.5 Shadowing

If two *forms* that *establish lexical bindings* with the same *name* N are textually nested, then references to N within the inner *form* refer to the *binding* established by the inner *form*; the inner *binding* for N *shadows* the outer *binding* for N. Outside the inner *form* but inside the outer one, references to N refer to the *binding* established by the outer *form*. For example:

```
(defun test (x z)
  (let ((z (* x 2)))
    (print z))
  z)
```

The *binding* of the variable `z` by **let** shadows the *parameter* binding for the function `test`. The reference to the variable `z` in the **print** *form* refers to the **let** binding. The reference to `z` at the end of the function `test` refers to the *parameter* named `z`.

Constructs that are lexically scoped act as if new names were generated for each *object* on each execution. Therefore, dynamic shadowing cannot occur. For example:

```
(defun contorted-example (f g x)
  (if (= x 0)
      (funcall f)
      (block here
         (+ 5 (contorted-example g
                                 #'(lambda () (return-from here 4))
                                 (- x 1))))))
```

Consider the call `(contorted-example nil nil 2)`. This produces `4`. During the course of execution, there are three calls to `contorted-example`, interleaved with two blocks:

```
(contorted-example nil nil 2)
  (block here1 ...)
    (contorted-example nil #'(lambda () (return-from here1 4)) 1)
      (block here2 ...)
        (contorted-example #'(lambda () (return-from here1 4))
                           #'(lambda () (return-from here2 4))
                           0)
            (funcall f)
                   where f =>  #'(lambda () (return-from here1 4))
                (return-from here1 4)
```

At the time the `funcall` is executed there are two **block** *exit points* outstanding, each apparently named `here`. The **return-from** *form* executed as a result of the `funcall` operation refers to the outer outstanding *exit point* (here1), not the inner one (here2). It refers to that *exit point* textually visible at the point of execution of **function** (here abbreviated by the `#'` syntax) that resulted in creation of the *function object* actually invoked by **funcall**.

If, in this example, one were to change the `(funcall f)` to `(funcall g)`, then the value of the call `(contorted-example nil nil 2)` would be `9`. The value would change because **funcall** would cause the execution of `(return-from here2 4)`, thereby causing a return from the inner *exit point* (here2). When that occurs, the value `4` is returned from the middle invocation of `contorted-example`, `5` is added to that to get `9`, and that value is returned from the outer block and the outermost call to `contorted-example`. The point is that the choice of *exit point* returned from has nothing to do with its being innermost or outermost; rather, it depends on the lexical environment that is packaged up with a *lambda expression* when **function** is executed.

3.1.6 Extent

`Contorted-example` works only because the *function* named by `f` is invoked during the *extent* of the *exit point*. Once the flow of execution has left the block, the *exit point* is *disestablished*. For example:

```
(defun invalid-example ()
  (let ((y (block here #'(lambda (z) (return-from here z)))))
    (if (numberp y) y (funcall y 5))))
```

One might expect the call `(invalid-example)` to produce `5` by the following incorrect reasoning: **let** binds `y` to the value of **block**; this value is a *function* resulting from the *lambda expression*. Because `y` is not a number, it is invoked on the value `5`. The **return-from** should then return this value from the *exit point* named `here`, thereby exiting from the block again and giving `y` the value `5` which, being a number, is then returned as the value of the call to `invalid-example`.

The argument fails only because *exit points* have *dynamic extent*. The argument is correct up to the execution of **return-from**. The execution of **return-from** should signal an error of *type* **control-error**, however, not because it cannot refer to the *exit point*, but because it does correctly refer to an *exit point* and that *exit point* has been *disestablished*.

A reference by name to a dynamic *exit point* binding such as a *catch tag* refers to the most recently *established binding* of that name that has not been *disestablished*. For example:

```
(defun fun1 (x)
  (catch 'trap (+ 3 (fun2 x))))
(defun fun2 (y)
  (catch 'trap (* 5 (fun3 y))))
(defun fun3 (z)
  (throw 'trap z))
```

Consider the call `(fun1 7)`. The result is `10`. At the time the **throw** is executed, there are two outstanding catchers with the name `trap`: one established within procedure `fun1`, and the other within procedure `fun2`. The latter is the more recent, and so the value `7` is returned from **catch** in `fun2`. Viewed from within `fun3`, the **catch** in `fun2` shadows the one in `fun1`. Had `fun2` been defined as

```
(defun fun2 (y)
  (catch 'snare (* 5 (fun3 y))))
```

then the two *exit points* would have different *names*, and therefore the one in `fun1` would not be shadowed. The result would then have been `7`.

3.1.7 Return Values

Ordinarily the result of calling a *function* is a single *object*. Sometimes, however, it is convenient for a function to compute several *objects* and return them.

In order to receive other than exactly one value from a *form*, one of several *special forms* or *macros* must be used to request those values. If a *form* produces *multiple values* which were not requested in this way, then the first value is given to the caller and all others are discarded; if the *form* produces zero values, then the caller receives **nil**

as a value.

The next figure lists some *operators* for receiving *multiple values*[2]. These *operators* can be used to specify one or more *forms* to *evaluate* and where to put the *values* returned by those *forms*.

```
multiple-value-bind  multiple-value-prog1  return-from
multiple-value-call  multiple-value-setq   throw
multiple-value-list  return
```

Figure 3-5. Some operators applicable to receiving multiple values

The *function* **values** can produce *multiple values*[2]. `(values)` returns zero values; `(values` *form*`)` returns the *primary value* returned by *form*; `(values` *form1 form2*`)` returns two values, the *primary value* of *form1* and the *primary value* of *form2*; and so on.

See **multiple-values-limit** and **values-list**.

3.2 Compilation

3.2.1 Compiler Terminology

The following terminology is used in this section.

The *compiler* is a utility that translates code into an *implementation-dependent* form that might be represented or executed efficiently. The term *compiler* refers to both of the *functions* **compile** and **compile-file**.

The term *compiled code* refers to *objects* representing compiled programs, such as *objects* constructed by **compile** or by **load** when *loading* a *compiled file*.

The term *implicit compilation* refers to *compilation* performed during *evaluation*.

The term *literal object* refers to a quoted *object* or a *self-evaluating object* or an *object* that is a substructure of such an *object*. A *constant variable* is not itself a *literal object*.

The term *coalesce* is defined as follows. Suppose `A` and `B` are two *literal constants* in the *source code*, and that `A′` and `B′` are the corresponding *objects* in the *compiled code*. If `A′` and `B′` are **eql** but `A` and `B` are not **eql**, then it is said that `A` and `B` have been coalesced by the compiler.

The term *minimal compilation* refers to actions the compiler must take at *compile time*. These actions are specified in Section 3.2.2 (Compilation Semantics).

The verb *process* refers to performing *minimal compilation*, determining the time of evaluation for a *form*, and possibly *evaluating* that *form* (if required).

The term *further compilation* refers to *implementation-dependent* compilation beyond *minimal compilation*. That is, *processing* does not imply complete compilation. Block compilation and generation of machine-specific instructions are examples of further compilation. Further compilation is permitted to take place at *run time*.

Four different *environments* relevant to compilation are distinguished: the *startup environment*, the *compilation environment*, the *evaluation environment*, and the *run-time environment*.

The *startup environment* is the *environment* of the *Lisp image* from which the *compiler* was invoked.

The *compilation environment* is maintained by the compiler and is used to hold definitions and declarations to be used internally by the compiler. Only those parts of a definition needed for correct compilation are saved. The *compilation environment* is used as the *environment argument* to macro expanders called by the compiler. It is unspecified whether a definition available in the *compilation environment* can be used in an *evaluation* initiated in the *startup environment* or *evaluation environment*.

The *evaluation environment* is a *run-time environment* in which macro expanders and code specified by **eval-when** to be evaluated are evaluated. All evaluations initiated by the *compiler* take place in the *evaluation environment*.

The *run-time environment* is the *environment* in which the program being compiled will be executed.

The *compilation environment* inherits from the *evaluation environment*, and the *compilation environment* and *evaluation environment* might be *identical*. The *evaluation environment* inherits from the *startup environment*, and the *startup environment* and *evaluation environment* might be *identical*.

The term *compile time* refers to the duration of time that the compiler is processing *source code*. At *compile time*, only the *compilation environment* and the *evaluation environment* are available.

The term *compile-time definition* refers to a definition in the *compilation environment*. For example, when compiling a file, the definition of a function might be retained in the *compilation environment* if it is declared **inline**. This definition might not be available in the *evaluation environment*.

The term *run time* refers to the duration of time that the loader is loading compiled code or compiled code is being executed. At run time, only the *run-time environment* is available.

The term *run-time definition* refers to a definition in the *run-time environment*.

The term *run-time compiler* refers to the *function* **compile** or *implicit compilation*, for which the compilation and run-time *environments* are maintained in the same *Lisp image*. Note that when the *run-time compiler* is used, the *run-time environment* and *startup environment* are the same.

3.2.2 Compilation Semantics

Conceptually, compilation is a process that traverses code, performs certain kinds of syntactic and semantic analyses using information (such as proclamations and *macro* definitions) present in the *compilation environment*, and produces equivalent, possibly more efficient code.

3.2.2.1 Compiler Macros

A *compiler macro* can be defined for a *name* that also names a *function* or *macro*. That is, it is possible for a *function name* to name both a *function* and a *compiler macro*.

A *function name* names a *compiler macro* if **compiler-macro-function** is *true* of the *function name* in the *lexical environment* in which it appears. Creating a *lexical binding* for the *function name* not only creates a new local *function* or *macro* definition, but also *shadows*[2] the *compiler macro*.

The *function* returned by **compiler-macro-function** is a *function* of two arguments, called the expansion function. To expand a *compiler macro*, the expansion function is invoked by calling the *macroexpand hook* with the expansion function as its first argument, the entire compiler macro *form* as its second argument, and the current compilation *environment* (or with the current lexical *environment*, if the *form* is being processed by something other than **compile-file**) as its third argument. The *macroexpand hook*, in turn, calls the expansion function with the *form* as its first argument and the *environment* as its second argument. The return value from the expansion function, which is passed through by the *macroexpand hook*, might either be the *same form*, or else a form that can, at the discretion of the *code* doing the expansion, be used in place of the original *form*.

`*macroexpand-hook*  compiler-macro-function  define-compiler-macro`

Figure 3-6. Defined names applicable to compiler macros

3.2.2.1.1 Purpose of Compiler Macros

The purpose of the *compiler macro* facility is to permit selective source code transformations as optimization advice to the *compiler*. When a *compound form* is being processed (as by the compiler), if the *operator* names a *compiler macro* then the *compiler macro function* may be invoked on the form, and the resulting expansion recursively processed in preference to performing the usual processing on the original *form* according to its normal interpretation as a *function form* or *macro form*.

A *compiler macro function*, like a *macro function*, is a *function* of two *arguments*: the entire call *form* and the *environment*. Unlike an ordinary *macro function*, a *compiler macro function* can decline to provide an expansion merely by returning a value that is the *same* as the original *form*. The consequences are undefined if a *compiler macro function* destructively modifies any part of its *form* argument.

The *form* passed to the compiler macro function can either be a *list* whose *car* is the function name, or a *list* whose *car* is **funcall** and whose *cadr* is a list `(function` *name*`)`; note that this affects destructuring of the form argument by the *compiler macro function*. **define-compiler-macro** arranges for destructuring of arguments to be performed correctly for both possible formats.

When **compile-file** chooses to expand a *top level form* that is a *compiler macro form*, the expansion is also treated as a *top level form* for the purposes of **eval-when** processing; see Section 3.2.3.1 (Processing of Top Level Forms).

3.2.2.1.2 Naming of Compiler Macros

Compiler macros may be defined for *function names* that name *macros* as well as *functions*.

Compiler macro definitions are strictly global. There is no provision for defining local *compiler macros* in the way that **macrolet** defines local *macros*. Lexical bindings of a function name shadow any compiler macro definition associated with the name as well as its global *function* or *macro* definition.

Note that the presence of a compiler macro definition does not affect the values returned by functions that access *function* definitions (e.g., **fboundp**) or *macro* definitions (e.g., **macroexpand**). Compiler macros are global, and the function **compiler-macro-function** is sufficient to resolve their interaction with other lexical and global definitions.

3.2.2.1.3 When Compiler Macros Are Used

The presence of a *compiler macro* definition for a *function* or *macro* indicates that it is desirable for the *compiler* to use the expansion of the *compiler macro* instead of the original *function form* or *macro form*. However, no language processor (compiler, evaluator, or other code walker) is ever required to actually invoke *compiler macro functions*, or to make use of the resulting expansion if it does invoke a *compiler macro function*.

When the *compiler* encounters a *form* during processing that represents a call to a *compiler macro name* (that is not declared **notinline**), the *compiler* might expand the *compiler macro*, and might use the expansion in place of the original *form*.

When **eval** encounters a *form* during processing that represents a call to a *compiler macro name* (that is not declared **notinline**), **eval** might expand the *compiler macro*, and might use the expansion in place of the original *form*.

There are two situations in which a *compiler macro* definition must not be applied by any language processor:

- The global function name binding associated with the compiler macro is shadowed by a lexical binding of the function name.
- The function name has been declared or proclaimed **notinline** and the call form appears within the scope of the declaration.

It is unspecified whether *compiler macros* are expanded or used in any other situations.

3.2.2.1.3.1 Notes about the Implementation of Compiler Macros

Although it is technically permissible, as described above, for **eval** to treat *compiler macros* in the same situations as *compiler* might, this is not necessarily a good idea in *interpreted implementations*.

Compiler macros exist for the purpose of trading compile-time speed for run-time speed. Programmers who write *compiler macros* tend to assume that the *compiler macros* can take more time than normal *functions* and *macros* in order to produce code which is especially optimal for use at run time. Since **eval** in an *interpreted implementation* might perform semantic analysis of the same form multiple times, it might be inefficient in general for the *implementation* to choose to call *compiler macros* on every such *evaluation*.

Nevertheless, the decision about what to do in these situations is left to each *implementation*.

3.2.2.2 Minimal Compilation

Minimal compilation is defined as follows:

- All *compiler macro* calls appearing in the *source code* being compiled are expanded, if at all, at compile time; they will not be expanded at run time.
- All *macro* and *symbol macro* calls appearing in the source code being compiled are expanded at compile time in such a way that they will not be expanded again at run time. **macrolet** and **symbol-macrolet** are effectively replaced by *forms* corresponding to their bodies in which calls to *macros* are replaced by their expansions.
- The first *argument* in a **load-time-value** *form* in *source code* processed by **compile** is *evaluated* at *compile time*; in *source code* processed by **compile-file**, the compiler arranges for it to be *evaluated* at *load time*. In either case, the result of the *evaluation* is remembered and used later as the value of the **load-time-value** *form* at *execution time*.

3.2.2.3 Semantic Constraints

All *conforming programs* must obey the following constraints, which are designed to minimize the observable differences between compiled and interpreted programs:

- Definitions of any referenced *macros* must be present in the *compilation environment*. Any *form* that is a *list* beginning with a *symbol* that does not name a *special operator* or a *macro* defined in the *compilation environment* is treated by the compiler as a function call.
- **Special** proclamations for *dynamic variables* must be made in the *compilation environment*. Any *binding* for which there is no **special** declaration or proclamation in the *compilation environment* is treated by the compiler as a *lexical binding*.
- The definition of a function that is defined and declared **inline** in the *compilation environment* must be the same at run time.
- Within a *function* named F, the compiler may (but is not required to) assume that an apparent recursive call to a *function* named F refers to the same definition of F, unless that function has been declared **notinline**. The consequences of redefining such a recursively defined *function* F while it is executing are undefined.
- A call within a file to a named function that is defined in the same file refers to that function, unless that

function has been declared **notinline**. The consequences are unspecified if functions are redefined individually at run time or multiply defined in the same file.

- The argument syntax and number of return values for all functions whose **ftype** is declared at compile time must remain the same at run time.
- *Constant variables* defined in the *compilation environment* must have a *similar* value at run time. A reference to a *constant variable* in *source code* is equivalent to a reference to a *literal object* that is the *value* of the *constant variable*.
- Type definitions made with **deftype** or **defstruct** in the *compilation environment* must retain the same definition at run time. Classes defined by **defclass** in the *compilation environment* must be defined at run time to have the same *superclasses* and same *metaclass*.

  This implies that *subtype*/*supertype* relationships of *type specifiers* must not change between *compile time* and *run time*.

- Type declarations present in the compilation *environment* must accurately describe the corresponding values at run time; otherwise, the consequences are undefined. It is permissible for an unknown *type* to appear in a declaration at compile time, though a warning might be signaled in such a case.
- Except in the situations explicitly listed above, a *function* defined in the *evaluation environment* is permitted to have a different definition or a different *signature* at run time, and the run-time definition prevails.

Conforming programs should not be written using any additional assumptions about consistency between the run-time *environment* and the startup, evaluation, and compilation *environments*.

Except where noted, when a compile-time and a run-time definition are different, one of the following occurs at run time:

- an error of *type* **error** is signaled
- the compile-time definition prevails
- the run-time definition prevails

If the *compiler* processes a *function form* whose *operator* is not defined at compile time, no error is signaled at compile time.

3.2.3 File Compilation

The *function* **compile-file** performs compilation of *forms* in a file following the rules specified in Section 3.2.2 (Compilation Semantics), and produces an output file that can be loaded by using **load**.

Normally, the *top level forms* appearing in a file compiled with **compile-file** are evaluated only when the resulting compiled file is loaded, and not when the file is compiled. However, it is typically the case that some forms in the file need to be evaluated at compile time so the remainder of the file can be read and compiled correctly.

The **eval-when** *special form* can be used to control whether a *top level form* is evaluated at compile time, load time, or both. It is possible to specify any of three situations with **eval-when**, denoted by the symbols `:compile-toplevel`, `:load-toplevel`, and `:execute`. For top level **eval-when** forms, `:compile-toplevel` specifies that the compiler must evaluate the body at compile time, and `:load-toplevel` specifies that the compiler must arrange to evaluate the body at load time. For non-top level **eval-when** forms, `:execute` specifies that the body must be executed in the run-time *environment*.

The behavior of this *form* can be more precisely understood in terms of a model of how **compile-file** processes forms in a file to be compiled. There are two processing modes, called "not-compile-time" and "compile-time-too".

Successive forms are read from the file by **compile-file** and processed in not-compile-time mode; in this mode, **compile-file** arranges for forms to be evaluated only at load time and not at compile time. When **compile-file** is in compile-time-too mode, forms are evaluated both at compile time and load time.

3.2.3.1 Processing of Top Level Forms

Processing of *top level forms* in the file compiler is defined as follows:

1. If the *form* is a *compiler macro form* (not disabled by a **notinline** *declaration*), the *implementation* might or might not choose to compute the *compiler macro expansion* of the *form* and, having performed the expansion, might or might not choose to process the result as a *top level form* in the same processing mode (compile-time-too or not-compile-time). If it declines to obtain or use the expansion, it must process the original *form*.
2. If the form is a *macro form*, its *macro expansion* is computed and processed as a *top level form* in the same processing mode (compile-time-too or not-compile-time).
3. If the form is a **progn** form, each of its body *forms* is sequentially processed as a *top level form* in the same processing mode.
4. If the form is a **locally**, **macrolet**, or **symbol-macrolet**, **compile-file** establishes the appropriate bindings and processes the body forms as *top level forms* with those bindings in effect in the same processing mode. (Note that this implies that the lexical *environment* in which *top level forms* are processed is not necessarily the *null lexical environment*.)
5. If the form is an **eval-when** form, it is handled according to the next figure.

   plus .5 fil

CT	LT	E	Mode	Action	New Mode
Yes	Yes	---	---	Process	compile-time-too
No	Yes	Yes	CTT	Process	compile-time-too
No	Yes	Yes	NCT	Process	not-compile-time
No	Yes	No	---	Process	not-compile-time
Yes	No	---	---	Evaluate	---
No	No	Yes	CTT	Evaluate	---
No	No	Yes	NCT	Discard	---
No	No	No	---	Discard	---

Figure 3-7. EVAL-WHEN processing

Column **CT** indicates whether `:compile-toplevel` is specified. Column **LT** indicates whether `:load-toplevel` is specified. Column **E** indicates whether `:execute` is specified. Column **Mode** indicates the processing mode; a dash (---) indicates that the processing mode is not relevant.

The **Action** column specifies one of three actions:

Process: process the body as *top level forms* in the specified mode.
Evaluate: evaluate the body in the dynamic execution context of the compiler, using the *evaluation environment* as the global environment and the *lexical environment* in which the **eval-when** appears.
Discard: ignore the *form*.

The **New Mode** column indicates the new processing mode. A dash (---) indicates the compiler remains in its current mode.

6. Otherwise, the form is a *top level form* that is not one of the special cases. In compile-time-too mode, the compiler first evaluates the form in the evaluation *environment* and then minimally compiles it. In not-compile-time mode, the *form* is simply minimally compiled. All *subforms* are treated as *non-top-level forms*.

   Note that *top level forms* are processed in the order in which they textually appear in the file and that each *top level form* read by the compiler is processed before the next is read. However, the order of processing (including macro expansion) of *subforms* that are not *top level forms* and the order of further compilation is unspecified as long as Common Lisp semantics are preserved.

eval-when forms cause compile-time evaluation only at top level. Both `:compile-toplevel` and `:load-toplevel` situation specifications are ignored for *non-top-level forms*. For *non-top-level forms*, an **eval-when** specifying the `:execute` situation is treated as an *implicit progn* including the *forms* in the body of the **eval-when** *form*; otherwise, the *forms* in the body are ignored.

3.2.3.1.1 Processing of Defining Macros

Defining *macros* (such as **defmacro** or **defvar**) appearing within a file being processed by **compile-file** normally have compile-time side effects which affect how subsequent *forms* in the same *file* are compiled. A convenient model for explaining how these side effects happen is that the defining macro expands into one or more **eval-when** *forms*, and that the calls which cause the compile-time side effects to happen appear in the body of an `(eval-when (:compile-toplevel) ...)` *form*.

The compile-time side effects may cause information about the definition to be stored differently than if the defining macro had been processed in the 'normal' way (either interpretively or by loading the compiled file).

In particular, the information stored by the defining *macros* at compile time might or might not be available to the interpreter (either during or after compilation), or during subsequent calls to the *compiler*. For example, the following code is nonportable because it assumes that the *compiler* stores the macro definition of `foo` where it is available to the interpreter:

```
(defmacro foo (x) '(car ,x))
(eval-when (:execute :compile-toplevel :load-toplevel)
  (print (foo '(a b c))))
```

A portable way to do the same thing would be to include the macro definition inside the **eval-when** *form*, as in:

```
(eval-when (:execute :compile-toplevel :load-toplevel)
  (defmacro foo (x) '(car ,x))
  (print (foo '(a b c))))
```

The next figure lists macros that make definitions available both in the compilation and run-time *environments*. It is not specified whether definitions made available in the *compilation environment* are available in the evaluation *environment*, nor is it specified whether they are available in subsequent compilation units or subsequent invocations of the compiler. As with **eval-when**, these compile-time side effects happen only when the defining macros appear at top level.

```
declaim                define-modify-macro   defsetf
defclass               define-setf-expander  defstruct
defconstant            defmacro              deftype
define-compiler-macro  defpackage            defvar
define-condition       defparameter
```

Figure 3-8. Defining Macros That Affect the Compile-Time Environment

3.2.3.1.2 Constraints on Macros and Compiler Macros

Except where explicitly stated otherwise, no *macro* defined in the Common Lisp standard produces an expansion that could cause any of the *subforms* of the *macro form* to be treated as *top level forms*. If an *implementation* also provides a *special operator* definition of a Common Lisp *macro*, the *special operator* definition must be semantically equivalent in this respect.

Compiler macro expansions must also have the same top level evaluation semantics as the *form* which they replace. This is of concern both to *conforming implementations* and to *conforming programs*.

3.2.4 Literal Objects in Compiled Files

The functions **eval** and **compile** are required to ensure that *literal objects* referenced within the resulting interpreted or compiled code objects are the *same* as the corresponding *objects* in the *source code*. **compile-file**, on the other hand, must produce a *compiled file* that, when loaded with **load**, constructs the *objects* defined by the *source code* and produces references to them.

In the case of **compile-file**, *objects* constructed by **load** of the *compiled file* cannot be spoken of as being the *same* as the *objects* constructed at compile time, because the *compiled file* may be loaded into a different *Lisp image* than the one in which it was compiled. This section defines the concept of *similarity* which relates *objects* in the *evaluation environment* to the corresponding *objects* in the *run-time environment*.

The constraints on *literal objects* described in this section apply only to **compile-file**; **eval** and **compile** do not copy or coalesce constants.

3.2.4.1 Externalizable Objects

The fact that the *file compiler* represents *literal objects* externally in a *compiled file* and must later reconstruct suitable equivalents of those *objects* when that *file* is loaded imposes a need for constraints on the nature of the *objects* that can be used as *literal objects* in *code* to be processed by the *file compiler*.

An *object* that can be used as a *literal object* in *code* to be processed by the *file compiler* is called an *externalizable object*.

We define that two *objects* are *similar* if they satisfy a two-place conceptual equivalence predicate (defined below), which is independent of the *Lisp image* so that the two *objects* in different *Lisp images* can be understood to be equivalent under this predicate. Further, by inspecting the definition of this conceptual predicate, the programmer can anticipate what aspects of an *object* are reliably preserved by *file compilation*.

The *file compiler* must cooperate with the *loader* in order to assure that in each case where an *externalizable object* is processed as a *literal object*, the *loader* will construct a *similar object*.

The set of *objects* that are *externalizable objects* are those for which the new conceptual term "*similar*" is defined, such that when a *compiled file* is *loaded*, an *object* can be constructed which can be shown to be *similar* to the original *object* which existed at the time the *file compiler* was operating.

3.2.4.2 Similarity of Literal Objects

3.2.4.2.1 Similarity of Aggregate Objects

Of the *types* over which *similarity* is defined, some are treated as aggregate objects. For these types, *similarity* is defined recursively. We say that an *object* of these types has certain "basic qualities" and to satisfy the *similarity* relationship, the values of the corresponding qualities of the two *objects* must also be similar.

3.2.4.2.2 Definition of Similarity

Two *objects* S (in *source code*) and C (in *compiled code*) are defined to be *similar* if and only if they are both of one of the *types* listed here (or defined by the *implementation*) and they both satisfy all additional requirements of *similarity* indicated for that *type*.

number
Two *numbers* S and C are *similar* if they are of the same *type* and represent the same mathematical value.

character
Two *simple characters* S and C are *similar* if they have *similar code attributes*.

Implementations providing additional, *implementation-defined attributes* must define whether and how *non-simple characters* can be regarded as *similar*.

symbol
Two *apparently uninterned symbols* S and C are *similar* if their *names* are *similar*.

Two *interned* symbols S and C are *similar* if their *names* are *similar*, and if either S is accessible in the *current package* at compile time and C is accessible in the *current package* at load time, or C is accessible in the *package* that is *similar* to the *home package* of S.

(Note that *similarity* of *symbols* is dependent on neither the *current readtable* nor how the *function* **read** would parse the *characters* in the *name* of the *symbol*.)

package
Two *packages* S and C are *similar* if their *names* are *similar*.

Note that although a *package object* is an *externalizable object*, the programmer is responsible for ensuring that the corresponding *package* is already in existence when code referencing it as a *literal object* is *loaded*. The *loader* finds the corresponding *package object* as if by calling **find-package** with that *name* as an *argument*. An error is signaled by the *loader* if no *package* exists at load time.

random-state
Two *random states* S and C are *similar* if S would always produce the same sequence of pseudo-random numbers as a *copy*[5] of C when given as the *random-state argument* to the *function* **random**, assuming equivalent *limit arguments* in each case.

(Note that since C has been processed by the *file compiler*, it cannot be used directly as an *argument* to **random** because **random** would perform a side effect.)

cons
Two *conses*, S and C, are *similar* if the *car*[2] of S is *similar* to the *car*[2] of C, and the *cdr*[2] of S is *similar* to the *cdr*[2] of C.

array
Two one-dimensional *arrays*, S and C, are *similar* if the *length* of S is *similar* to the *length* of C, the *actual array element type* of S is *similar* to the *actual array element type* of C, and each *active element* of S is *similar* to the corresponding *element* of C.

Two *arrays* of *rank* other than one, S and C, are *similar* if the *rank* of S is *similar* to the *rank* of C, each *dimension*[1] of S is *similar* to the corresponding *dimension*[1] of C, the *actual array element type* of S is *similar* to the *actual array element type* of C, and each *element* of S is *similar* to the corresponding *element* of C.

In addition, if S is a *simple array*, then C must also be a *simple array*. If S is a *displaced array*, has a *fill pointer*, or is *actually adjustable*, C is permitted to lack any or all of these qualities.

hash-table

Two *hash tables* S and C are *similar* if they meet the following three requirements:

1. They both have the same test (e.g., they are both **eql** *hash tables*).
2. There is a unique one-to-one correspondence between the keys of the two *hash tables*, such that the corresponding keys are *similar*.
3. For all keys, the values associated with two corresponding keys are *similar*.

If there is more than one possible one-to-one correspondence between the keys of S and C, the consequences are unspecified. A *conforming program* cannot use a table such as S as an *externalizable constant*.

pathname

Two *pathnames* S and C are *similar* if all corresponding *pathname components* are *similar*.

function

Functions are not *externalizable objects*.

structure-object and **standard-object**

A general-purpose concept of *similarity* does not exist for *structures* and *standard objects*. However, a *conforming program* is permitted to define a **make-load-form** *method* for any *class* K defined by that *program* that is a *subclass* of either **structure-object** or **standard-object**. The effect of such a *method* is to define that an *object* S of *type* K in *source code* is *similar* to an *object* C of *type* K in *compiled code* if C was constructed from *code* produced by calling **make-load-form** on S.

3.2.4.3 Extensions to Similarity Rules

Some *objects*, such as *streams*, **readtables**, and **methods** are not *externalizable objects* under the definition of similarity given above. That is, such *objects* may not portably appear as *literal objects* in *code* to be processed by the *file compiler*.

An *implementation* is permitted to extend the rules of similarity, so that other kinds of *objects* are *externalizable objects* for that *implementation*.

If for some kind of *object*, *similarity* is neither defined by this specification nor by the *implementation*, then the *file compiler* must signal an error upon encountering such an *object* as a *literal constant*.

3.2.4.4 Additional Constraints on Externalizable Objects

If two *literal objects* appearing in the source code for a single file processed with the *file compiler* are the *identical*, the corresponding *objects* in the *compiled code* must also be the *identical*. With the exception of *symbols* and *packages*, any two *literal objects* in *code* being processed by the *file compiler* may be *coalesced* if and only if they are *similar*; if they are either both *symbols* or both *packages*, they may only be *coalesced* if and only if they are *identical*.

Objects containing circular references can be *externalizable objects*. The *file compiler* is required to preserve **eql**ness of substructures within a *file*. Preserving **eql**ness means that subobjects that are the *same* in the *source code* must be the *same* in the corresponding *compiled code*.

In addition, the following are constraints on the handling of *literal objects* by the *file compiler*:

array: If an *array* in the source code is a *simple array*, then the corresponding *array* in the compiled code will also be a *simple array*. If an *array* in the source code is displaced, has a *fill pointer*, or is *actually adjustable*, the corresponding *array* in the compiled code might lack any or all of these qualities. If an *array* in the source code has a fill pointer, then the corresponding *array* in the compiled code might be only the size implied by the fill pointer.
packages: The loader is required to find the corresponding *package object* as if by calling **find-package** with the package name as an argument. An error of *type* **package-error** is signaled if no *package* of that name exists at load time.
random-state: A constant *random state* object cannot be used as the state argument to the *function* **random** because **random** modifies this data structure.
structure, standard-object: *Objects* of *type* **structure-object** and **standard-object** may appear in compiled constants if there is an appropriate **make-load-form** method defined for that *type*.

The *file compiler* calls **make-load-form** on any *object* that is referenced as a *literal object* if the *object* is a *generalized instance* of **standard-object**, **structure-object**, **condition**, or any of a (possibly empty) *implementation-dependent* set of other *classes*. The *file compiler* only calls **make-load-form** once for any given *object* within a single *file*.

symbol: In order to guarantee that *compiled files* can be *loaded* correctly, users must ensure that the *packages* referenced in those *files* are defined consistently at compile time and load time. *Conforming programs* must satisfy the following requirements:

1. The *current package* when a *top level form* in the *file* is processed by **compile-file** must be the same as the *current package* when the *code* corresponding to that *top level form* in the *compiled file* is executed by **load**. In particular:
   a. Any *top level form* in a *file* that alters the *current package* must change it to a *package* of the same *name* both at compile time and at load time.
   b. If the first *non-atomic top level form* in the *file* is not an **in-package** *form*, then the *current package* at the time **load** is called must be a *package* with the same *name* as the package that was the *current package* at the time **compile-file** was called.
2. For all *symbols* appearing lexically within a *top level form* that were *accessible* in the *package* that was the *current package* during processing of that *top level form* at compile time, but whose *home package* was another *package*, at load time there must be a *symbol* with the same *name* that is *accessible* in both the load-time *current package* and in the *package* with the same *name* as the compile-time *home package*.
3. For all *symbols* represented in the *compiled file* that were *external symbols* in their *home package* at compile time, there must be a *symbol* with the same *name* that is an *external symbol* in the *package* with the same *name* at load time.

If any of these conditions do not hold, the *package* in which the *loader* looks for the affected *symbols* is unspecified. *Implementations* are permitted to signal an error or to define this behavior.

3.2.5 Exceptional Situations in the Compiler

compile and **compile-file** are permitted to signal errors and warnings, including errors due to compile-time processing of `(eval-when (:compile-toplevel) ...)` forms, macro expansion, and conditions signaled by the compiler itself.

Conditions of *type* **error** might be signaled by the compiler in situations where the compilation cannot proceed without intervention.

In addition to situations for which the standard specifies that *conditions* of *type* **warning** must or might be signaled, warnings might be signaled in situations where the compiler can determine that the consequences are undefined or that a run-time error will be signaled. Examples of this situation are as follows: violating type declarations, altering or assigning the value of a constant defined with **defconstant**, calling built-in Lisp functions with a wrong number of arguments or malformed keyword argument lists, and using unrecognized declaration specifiers.

The compiler is permitted to issue warnings about matters of programming style as conditions of *type* **style-warning**. Examples of this situation are as follows: redefining a function using a different argument list, calling a function with a wrong number of arguments, not declaring **ignore** of a local variable that is not referenced, and referencing a variable declared **ignore**.

Both **compile** and **compile-file** are permitted (but not required) to *establish* a *handler* for *conditions* of *type* **error**. For example, they might signal a warning, and restart compilation from some *implementation-dependent* point in order to let the compilation proceed without manual intervention.

Both **compile** and **compile-file** return three values, the second two indicating whether the source code being compiled contained errors and whether style warnings were issued.

Some warnings might be deferred until the end of compilation. See **with-compilation-unit**.

3.3 Declarations

Declarations provide a way of specifying information for use by program processors, such as the evaluator or the compiler.

Local declarations can be embedded in executable code using **declare**. *Global declarations*, or *proclamations*, are established by **proclaim** or **declaim**.

The **the** *special form* provides a shorthand notation for making a *local declaration* about the *type* of the *value* of a given *form*.

The consequences are undefined if a program violates a *declaration* or a *proclamation*.

3.3.1 Minimal Declaration Processing Requirements

In general, an *implementation* is free to ignore *declaration specifiers* except for the **declaration**, **notinline**, **safety**, and **special** *declaration specifiers*.

A **declaration** *declaration* must suppress warnings about unrecognized *declarations* of the kind that it declares. If an *implementation* does not produce warnings about unrecognized declarations, it may safely ignore this *declaration*.

A **notinline** *declaration* must be recognized by any *implementation* that supports inline functions or *compiler macros* in order to disable those facilities. An *implementation* that does not use inline functions or *compiler macros* may safely ignore this *declaration*.

A **safety** *declaration* that increases the current safety level must always be recognized. An *implementation* that always processes code as if safety were high may safely ignore this *declaration*.

A **special** *declaration* must be processed by all *implementations*.

3.3.2 Declaration Specifiers

A *declaration specifier* is an *expression* that can appear at top level of a **declare** expression or a **declaim** form, or as the argument to **proclaim**. It is a *list* whose *car* is a *declaration identifier*, and whose *cdr* is data interpreted according to rules specific to the *declaration identifier*.

3.3.3 Declaration Identifiers

The next figure shows a list of all *declaration identifiers* defined by this standard.

```
declaration     ignore     special
dynamic-extent  inline     type
ftype           notinline
ignorable       optimize
```

Figure 3-9. Common Lisp Declaration Identifiers

An implementation is free to support other (*implementation-defined*) *declaration identifiers* as well. A warning might be issued if a *declaration identifier* is not among those defined above, is not defined by the *implementation*, is not a *type name*, and has not been declared in a **declaration** *proclamation*.

3.3.3.1 Shorthand notation for Type Declarations

A *type specifier* can be used as a *declaration identifier*. (*type-specifier var**) is taken as shorthand for (`type` *type-specifier var**).

3.3.4 Declaration Scope

Declarations can be divided into two kinds: those that apply to the *bindings* of *variables* or *functions*; and those that do not apply to *bindings*.

A *declaration* that appears at the head of a binding *form* and applies to a *variable* or *function binding* made by that *form* is called a *bound declaration*; such a *declaration* affects both the *binding* and any references within the *scope* of the *declaration*.

Declarations that are not *bound declarations* are called *free declarations*.

A *free declaration* in a *form* F1 that applies to a *binding* for a *name* N *established* by some *form* F2 of which F1 is a *subform* affects only references to N within F1; it does not to apply to other references to N outside of F1, nor does it affect the manner in which the *binding* of N by F2 is *established*.

Declarations that do not apply to *bindings* can only appear as *free declarations*.

The *scope* of a *bound declaration* is the same as the *lexical scope* of the *binding* to which it applies; for *special variables*, this means the *scope* that the *binding* would have had had it been a *lexical binding*.

Unless explicitly stated otherwise, the *scope* of a *free declaration* includes only the body *subforms* of the *form* at whose head it appears, and no other *subforms*. The *scope* of *free declarations* specifically does not include *initialization forms* for *bindings* established by the *form* containing the *declarations*.

Some *iteration forms* include step, end-test, or result *subforms* that are also included in the *scope* of *declarations* that appear in the *iteration form*. Specifically, the *iteration forms* and *subforms* involved are:

- **do**, **do***: *step-forms*, *end-test-form*, and *result-forms*.
- **dolist**, **dotimes**: *result-form*
- **do-all-symbols**, **do-external-symbols**, **do-symbols**: *result-form*

3.3.4.1 Examples of Declaration Scope

Here is an example illustrating the *scope* of *bound declarations*.

```
 (let ((x 1))                ;[1] 1st occurrence of x
   (declare (special x))     ;[2] 2nd occurrence of x
   (let ((x 2))              ;[3] 3rd occurrence of x
     (let ((old-x x)         ;[4] 4th occurrence of x
           (x 3))            ;[5] 5th occurrence of x
       (declare (special x)) ;[6] 6th occurrence of x
       (list old-x x))))     ;[7] 7th occurrence of x
=>  (2 3)
```

The first occurrence of `x` *establishes* a *dynamic binding* of `x` because of the **special** *declaration* for `x` in the second line. The third occurrence of `x` *establishes* a *lexical binding* of `x` (because there is no **special** *declaration* in the corresponding **let** *form*). The fourth occurrence of `x` *x* is a reference to the *lexical binding* of `x` established in the third line. The fifth occurrence of `x` *establishes* a *dynamic binding* of *x* for the body of the **let** *form* that begins on that line because of the **special** *declaration* for `x` in the sixth line. The reference to `x` in the fourth line is not affected by the **special** *declaration* in the sixth line because that reference is not within the "would-be *lexical scope*" of the *variable* `x` in the fifth line. The reference to `x` in the seventh line is a reference to the *dynamic binding* of *x established* in the fifth line.

Here is another example, to illustrate the *scope* of a *free declaration*. In the following:

```
 (lambda (&optional (x (foo 1))) ;[1]
   (declare (notinline foo))     ;[2]
   (foo x))                      ;[3]
```

the *call* to `foo` in the first line might be compiled inline even though the *call* to `foo` in the third line must not be. This is because the **notinline** *declaration* for `foo` in the second line applies only to the body on the third line. In order to suppress inlining for both *calls*, one might write:

```
 (locally (declare (notinline foo)) ;[1]
   (lambda (&optional (x (foo 1)))  ;[2]
     (foo x)))                      ;[3]
```

or, alternatively:

```
 (lambda (&optional                                ;[1]
           (x (locally (declare (notinline foo)) ;[2]
                (foo 1))))                       ;[3]
   (declare (notinline foo))                      ;[4]
   (foo x))                                       ;[5]
```

Finally, here is an example that shows the *scope* of *declarations* in an *iteration form*.

```
 (let ((x  1))                    ;[1]
   (declare (special x))          ;[2]
     (let ((x 2))                 ;[3]
       (dotimes (i x x)           ;[4]
         (declare (special x))))  ;[5]
=>  1
```

In this example, the first reference to `x` on the fourth line is to the *lexical binding* of `x` established on the third line. However, the second occurrence of `x` on the fourth line lies within the *scope* of the *free declaration* on the fifth line (because this is the *result-form* of the **dotimes**) and therefore refers to the *dynamic binding* of `x`.

3.4 Lambda Lists

A *lambda list* is a *list* that specifies a set of *parameters* (sometimes called *lambda variables*) and a protocol for receiving *values* for those *parameters*.

There are several kinds of *lambda lists*.

Context	Kind of Lambda List
defun form	ordinary lambda list
defmacro form	macro lambda list
lambda expression	ordinary lambda list
flet local function definition	ordinary lambda list
labels local function definition	ordinary lambda list
handler-case clause specification	ordinary lambda list
restart-case clause specification	ordinary lambda list
macrolet local macro definition	macro lambda list
define-method-combination	ordinary lambda list
define-method-combination :arguments option	define-method-combination arguments lambda list
defstruct :constructor option	boa lambda list
defgeneric form	generic function lambda list
defgeneric method clause	specialized lambda list
defmethod form	specialized lambda list
defsetf form	defsetf lambda list
define-setf-expander form	macro lambda list
deftype form	deftype lambda list
destructuring-bind form	destructuring lambda list
define-compiler-macro form	macro lambda list
define-modify-macro form	define-modify-macro lambda list

Figure 3-10. What Kind of Lambda Lists to Use

The next figure lists some *defined names* that are applicable to *lambda lists*.

lambda-list-keywords	lambda-parameters-limit

Figure 3-11. Defined names applicable to lambda lists

3.4.1 Ordinary Lambda Lists

An *ordinary lambda list* is used to describe how a set of *arguments* is received by an *ordinary function*. The *defined names* in the next figure are those which use *ordinary lambda lists*:

define-method-combination	handler-case	restart-case
defun	labels	
flet	lambda	

Figure 3-12. Standardized Operators that use Ordinary Lambda Lists

An *ordinary lambda list* can contain the *lambda list keywords* shown in the next figure.

&allow-other-keys	&key	&rest
&aux	&optional	

Figure 3-13. Lambda List Keywords used by Ordinary Lambda Lists

Each *element* of a *lambda list* is either a parameter specifier or a *lambda list keyword*. Implementations are free to provide additional *lambda list keywords*. For a list of all *lambda list keywords* used by the implementation, see **lambda-list-keywords**.

The syntax for *ordinary lambda lists* is as follows:

```
lambda-list::= (var*
                [&optional {var | (var [init-form [supplied-p-parameter]])}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
                [&aux {var | (var [init-form])}*])
```

A *var* or *supplied-p-parameter* must be a *symbol* that is not the name of a *constant variable*.

An *init-form* can be any *form*. Whenever any *init-form* is evaluated for any parameter specifier, that *form* may refer to any parameter variable to the left of the specifier in which the *init-form* appears, including any *supplied-p-parameter* variables, and may rely on the fact that no other parameter variable has yet been bound (including its own parameter variable).

A *keyword-name* can be any *symbol*, but by convention is normally a *keyword*[1]; all *standardized functions* follow that convention.

An *ordinary lambda list* has five parts, any or all of which may be empty. For information about the treatment of argument mismatches, see Section 3.5 (Error Checking in Function Calls).

3.4.1.1 Specifiers for the required parameters

These are all the parameter specifiers up to the first *lambda list keyword*; if there are no *lambda list keywords*, then all the specifiers are for required parameters. Each required parameter is specified by a parameter variable *var*. *var* is bound as a lexical variable unless it is declared **special**.

If there are `n` required parameters (`n` may be zero), there must be at least `n` passed arguments, and the required parameters are bound to the first `n` passed arguments; see Section 3.5 (Error Checking in Function Calls). The other parameters are then processed using any remaining arguments.

3.4.1.2 Specifiers for optional parameters

If `&optional` is present, the optional parameter specifiers are those following `&optional` up to the next *lambda list keyword* or the end of the list. If optional parameters are specified, then each one is processed as follows. If any unprocessed arguments remain, then the parameter variable *var* is bound to the next remaining argument, just as for a required parameter. If no arguments remain, however, then *init-form* is evaluated, and the parameter variable is bound to the resulting value (or to **nil** if no *init-form* appears in the parameter specifier). If another variable name *supplied-p-parameter* appears in the specifier, it is bound to *true* if an argument had been available, and to *false* if no argument remained (and therefore *init-form* had to be evaluated). *Supplied-p-parameter* is bound not to an argument but to a value indicating whether or not an argument had been supplied for the corresponding *var*.

3.4.1.3 A specifier for a rest parameter

`&rest`, if present, must be followed by a single *rest parameter* specifier, which in turn must be followed by another *lambda list keyword* or the end of the *lambda list*. After all optional parameter specifiers have been processed, then there may or may not be a *rest parameter*. If there is a *rest parameter*, it is bound to a *list* of all as-yet-unprocessed arguments. If no unprocessed arguments remain, the *rest parameter* is bound to the *empty list*. If there is no *rest parameter* and there are no *keyword parameters*, then an error should be signaled if any unprocessed arguments remain; see Section 3.5 (Error Checking in Function Calls). The value of a *rest parameter* is permitted, but not required, to share structure with the last argument to **apply**.

3.4.1.4 Specifiers for keyword parameters

If `&key` is present, all specifiers up to the next *lambda list keyword* or the end of the *list* are keyword parameter specifiers. When keyword parameters are processed, the same arguments are processed that would be made into a *list* for a *rest parameter*. It is permitted to specify both `&rest` and `&key`. In this case the remaining arguments are used for both purposes; that is, all remaining arguments are made into a *list* for the *rest parameter*, and are also processed for the `&key` parameters. If `&key` is specified, there must remain an even number of arguments; see Section 3.5.1.6 (Odd Number of Keyword Arguments). These arguments are considered as pairs, the first argument in each pair being interpreted as a name and the second as the corresponding value. The first *object* of each pair must be a *symbol*; see Section 3.5.1.5 (Invalid Keyword Arguments). The keyword parameter specifiers may optionally be followed by the *lambda list keyword* `&allow-other-keys`.

In each keyword parameter specifier must be a name *var* for the parameter variable. If the *var* appears alone or in a `(`*var init-form*`)` combination, the keyword name used when matching *arguments* to *parameters* is a *symbol* in the `KEYWORD` package whose *name* is the *same* (under **string=**) as *var*'s. If the notation `((`*keyword-name var*`)` *init-form*`)` is used, then the keyword name used to match *arguments* to *parameters* is *keyword-name*, which may be a *symbol* in any *package*. (Of course, if it is not a *symbol* in the `KEYWORD` package, it does not necessarily self-evaluate, so care must be taken when calling the function to make sure that normal evaluation still yields the keyword name.) Thus

```
(defun foo (&key radix (type 'integer)) ...)
```

means exactly the same as

```
(defun foo (&key ((:radix radix)) ((:type type) 'integer)) ...)
```

The keyword parameter specifiers are, like all parameter specifiers, effectively processed from left to right. For each keyword parameter specifier, if there is an argument pair whose name matches that specifier's name (that is, the names are **eq**), then the parameter variable for that specifier is bound to the second item (the value) of that argument pair. If more than one such argument pair matches, the leftmost argument pair is used. If no such argument pair exists, then the *init-form* for that specifier is evaluated and the parameter variable is bound to that value (or to **nil** if no *init-form* was specified). *supplied-p-parameter* is treated as for `&optional` parameters: it is bound to *true* if there was a matching argument pair, and to *false* otherwise.

Unless keyword argument checking is suppressed, an argument pair must a name matched by a parameter specifier; see Section 3.5.1.4 (Unrecognized Keyword Arguments).

If keyword argument checking is suppressed, then it is permitted for an argument pair to match no parameter specifier, and the argument pair is ignored, but such an argument pair is accessible through the *rest parameter* if one was supplied. The purpose of these mechanisms is to allow sharing of argument lists among several *lambda expressions* and to allow either the caller or the called *lambda expression* to specify that such sharing may be taking place.

Note that if `&key` is present, a keyword argument of `:allow-other-keys` is always permitted---regardless of whether the associated value is *true* or *false*. However, if the value is *false*, other non-matching keywords are not tolerated (unless `&allow-other-keys` was used).

Furthermore, if the receiving argument list specifies a regular argument which would be flagged by `:allow-other-keys`, then `:allow-other-keys` has both its special-cased meaning (identifying whether additional keywords are permitted) and its normal meaning (data flow into the function in question).

3.4.1.4.1 Suppressing Keyword Argument Checking

If `&allow-other-keys` was specified in the *lambda list* of a *function*, *keyword*[2] *argument* checking is suppressed in calls to that *function*.

If the `:allow-other-keys` *argument* is *true* in a call to a *function*, *keyword*[2] *argument* checking is suppressed in that call.

The `:allow-other-keys` *argument* is permissible in all situations involving *keyword*[2] *arguments*, even when its associated *value* is *false*.

3.4.1.4.1.1 Examples of Suppressing Keyword Argument Checking

```
;;; The caller can supply :ALLOW-OTHER-KEYS T to suppress checking.
 ((lambda (&key x) x) :x 1 :y 2 :allow-other-keys t) =>  1
;;; The callee can use &ALLOW-OTHER-KEYS to suppress checking.
 ((lambda (&key x &allow-other-keys) x) :x 1 :y 2) =>  1
;;; :ALLOW-OTHER-KEYS NIL is always permitted.
 ((lambda (&key) t) :allow-other-keys nil) =>  T
;;; As with other keyword arguments, only the left-most pair
;;; named :ALLOW-OTHER-KEYS has any effect.
 ((lambda (&key x) x)
  :x 1 :y 2 :allow-other-keys t :allow-other-keys nil)
=>  1
;;; Only the left-most pair named :ALLOW-OTHER-KEYS has any effect,
;;; so in safe code this signals a PROGRAM-ERROR (and might enter the
;;; debugger).  In unsafe code, the consequences are undefined.
 ((lambda (&key x) x)                   ;This call is not valid
  :x 1 :y 2 :allow-other-keys nil :allow-other-keys t)
```

3.4.1.5 Specifiers for &aux variables

These are not really parameters. If the *lambda list keyword* `&aux` is present, all specifiers after it are auxiliary variable specifiers. After all parameter specifiers have been processed, the auxiliary variable specifiers (those following `&aux`) are processed from left to right. For each one, *init-form* is evaluated and *var* is bound to that value (or to **nil** if no *init-form* was specified). `&aux` variable processing is analogous to **let*** processing.

```
 (lambda (x y &aux (a (car x)) (b 2) c) (list x y a b c))
    ==  (lambda (x y) (let* ((a (car x)) (b 2) c) (list x y a b c)))
```

3.4.1.6 Examples of Ordinary Lambda Lists

Here are some examples involving *optional parameters* and *rest parameters*:

```
 ((lambda (a b) (+ a (* b 3))) 4 5) =>  19
 ((lambda (a &optional (b 2)) (+ a (* b 3))) 4 5) =>  19
 ((lambda (a &optional (b 2)) (+ a (* b 3))) 4) =>  10
 ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)))
=>  (2 NIL 3 NIL NIL)
 ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 6)
=>  (6 T 3 NIL NIL)
 ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 6 3)
=>  (6 T 3 T NIL)
 ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x)) 6 3 8)
=>  (6 T 3 T (8))
 ((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
  6 3 8 9 10 11)
=>  (6 t 3 t (8 9 10 11))
```

Here are some examples involving *keyword parameters*:

```
 ((lambda (a b &key c d) (list a b c d)) 1 2) =>  (1 2 NIL NIL)
 ((lambda (a b &key c d) (list a b c d)) 1 2 :c 6) =>  (1 2 6 NIL)
 ((lambda (a b &key c d) (list a b c d)) 1 2 :d 8) =>  (1 2 NIL 8)
 ((lambda (a b &key c d) (list a b c d)) 1 2 :c 6 :d 8) =>  (1 2 6 8)
 ((lambda (a b &key c d) (list a b c d)) 1 2 :d 8 :c 6) =>  (1 2 6 8)
 ((lambda (a b &key c d) (list a b c d)) :a 1 :d 8 :c 6) =>  (:a 1 6 8)
 ((lambda (a b &key c d) (list a b c d)) :a :b :c :d) =>  (:a :b :d NIL)
 ((lambda (a b &key ((:sea c)) d) (list a b c d)) 1 2 :sea 6) =>  (1 2 6 NIL)
 ((lambda (a b &key ((c c)) d) (list a b c d)) 1 2 'c 6) =>  (1 2 6 NIL)
```

Here are some examples involving *optional parameters*, *rest parameters*, and *keyword parameters* together:

```
 ((lambda (a &optional (b 3) &rest x &key c (d a))
    (list a b c d x)) 1)
=>  (1 3 NIL 1 ())
 ((lambda (a &optional (b 3) &rest x &key c (d a))
    (list a b c d x)) 1 2)
=>  (1 2 NIL 1 ())
 ((lambda (a &optional (b 3) &rest x &key c (d a))
    (list a b c d x)) :c 7)
=>  (:c 7 NIL :c ())
 ((lambda (a &optional (b 3) &rest x &key c (d a))
    (list a b c d x)) 1 6 :c 7)
=>  (1 6 7 1 (:c 7))
 ((lambda (a &optional (b 3) &rest x &key c (d a))
    (list a b c d x)) 1 6 :d 8)
=>  (1 6 NIL 8 (:d 8))
 ((lambda (a &optional (b 3) &rest x &key c (d a))
    (list a b c d x)) 1 6 :d 8 :c 9 :d 10)
=>  (1 6 9 8 (:d 8 :c 9 :d 10))
```

As an example of the use of `&allow-other-keys` and `:allow-other-keys`, consider a *function* that takes two named arguments of its own and also accepts additional named arguments to be passed to **make-array**:

```
 (defun array-of-strings (str dims &rest named-pairs
                          &key (start 0) end &allow-other-keys)
   (apply #'make-array dims
          :initial-element (subseq str start end)
          :allow-other-keys t
          named-pairs))
```

This *function* takes a *string* and dimensioning information and returns an *array* of the specified dimensions, each of whose elements is the specified *string*. However, `:start` and `:end` named arguments may be used to specify that a substring of the given *string* should be used. In addition, the presence of `&allow-other-keys` in the *lambda list* indicates that the caller may supply additional named arguments; the *rest parameter* provides access to them. These additional named arguments are passed to **make-array**. The *function* **make-array** normally does not allow the named arguments `:start` and `:end` to be used, and an error should be signaled if such named arguments are supplied to **make-array**. However, the presence in the call to **make-array** of the named argument `:allow-other-keys` with a *true* value causes any extraneous named arguments, including `:start` and `:end`, to be acceptable and ignored.

3.4.2 Generic Function Lambda Lists

A *generic function lambda list* is used to describe the overall shape of the argument list to be accepted by a *generic function*. Individual *method signatures* might contribute additional *keyword parameters* to the *lambda list* of the *effective method*.

A *generic function lambda list* is used by **defgeneric**.

A *generic function lambda list* has the following syntax:

```
lambda-list::= (var*
                [&optional {var | (var)}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)})}* [&allow-other-keys]])
```

A *generic function lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &optional
&key               &rest
```

Figure 3-14. Lambda List Keywords used by Generic Function Lambda Lists

A *generic function lambda list* differs from an *ordinary lambda list* in the following ways:

Required arguments
Zero or more *required parameters* must be specified.

Optional and keyword arguments
Optional parameters and *keyword parameters* may not have default initial value forms nor use supplied-p parameters.

Use of `&aux`
The use of `&aux` is not allowed.

3.4.3 Specialized Lambda Lists

A *specialized lambda list* is used to *specialize* a *method* for a particular *signature* and to describe how *arguments* matching that *signature* are received by the *method*. The *defined names* in the next figure use *specialized lambda lists* in some way; see the dictionary entry for each for information about how.

```
defmethod  defgeneric
```

Figure 3-15. Standardized Operators that use Specialized Lambda Lists

A *specialized lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&aux               &optional
```

Figure 3-16. Lambda List Keywords used by Specialized Lambda Lists

A *specialized lambda list* is syntactically the same as an *ordinary lambda list* except that each *required parameter* may optionally be associated with a *class* or *object* for which that *parameter* is *specialized*.

```
lambda-list::= ({var | (var [specializer])}*
                [&optional {var | (var [init-form [supplied-p-parameter]])}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
                [&aux {var | (var [init-form])}*])
```

3.4.4 Macro Lambda Lists

A *macro lambda list* is used in describing *macros* defined by the *operators* in the next figure.

```
define-compiler-macro  defmacro  macrolet
define-setf-expander
```

Figure 3-17. Operators that use Macro Lambda Lists

With the additional restriction that an *environment parameter* may appear only once (at any of the positions indicated), a *macro lambda list* has the following syntax:

```
reqvars::= var*

optvars::= [&optional {var | (var [init-form [supplied-p-parameter]])}*]

restvar::= [{&rest | &body} var]

keyvars::= [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}*
            [&allow-other-keys]]

auxvars::= [&aux {var | (var [init-form])}*]

envvar::= [&environment var]

wholevar::= [&whole var]

lambda-list::= (wholevar envvar  reqvars envvar  optvars envvar
                restvar envvar  keyvars envvar  auxvars envvar) |
               (wholevar envvar  reqvars envvar  optvars envvar .  var)

pattern::= (wholevar reqvars optvars restvar keyvars auxvars) |
           (wholevar reqvars optvars . var)
```

A *macro lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &environment  &rest
&aux               &key          &whole
&body              &optional
```

Figure 3-18. Lambda List Keywords used by Macro Lambda Lists

Optional parameters (introduced by `&optional`) and *keyword parameters* (introduced by `&key`) can be supplied in a *macro lambda list*, just as in an *ordinary lambda list*. Both may contain default initialization forms and *supplied-p parameters*.

`&body` is identical in function to `&rest`, but it can be used to inform certain output-formatting and editing functions that the remainder of the *form* is treated as a body, and should be indented accordingly. Only one of `&body` or `&rest` can be used at any particular level; see Section 3.4.4.1 (Destructuring by Lambda Lists). `&body` can appear at any level of a *macro lambda list*; for details, see Section 3.4.4.1 (Destructuring by Lambda Lists).

`&whole` is followed by a single variable that is bound to the entire macro-call form; this is the value that the *macro function* receives as its first argument. If `&whole` and a following variable appear, they must appear first in *lambda-list*, before any other parameter or *lambda list keyword*. `&whole` can appear at any level of a *macro lambda list*. At inner levels, the `&whole` variable is bound to the corresponding part of the argument, as with `&rest`, but unlike `&rest`, other arguments are also allowed. The use of `&whole` does not affect the pattern of arguments specified.

`&environment` is followed by a single variable that is bound to an *environment* representing the *lexical environment* in which the macro call is to be interpreted. This *environment* should be used with **macro-function**, **get-setf-expansion**, **compiler-macro-function**, and **macroexpand** (for example) in computing the expansion of the macro, to ensure that any *lexical bindings* or definitions established in the *compilation environment* are taken into account. `&environment` can only appear at the top level of a *macro lambda list*, and can only appear once, but can appear anywhere in that list; the `&environment` *parameter* is *bound* along with `&whole` before any

other *variables* in the *lambda list*, regardless of where `&environment` appears in the *lambda list*. The *object* that is bound to the *environment parameter* has *dynamic extent*.

Destructuring allows a *macro lambda list* to express the structure of a macro call syntax. If no *lambda list keywords* appear, then the *macro lambda list* is a *tree* containing parameter names at the leaves. The pattern and the *macro form* must have compatible *tree structure*; that is, their *tree structure* must be equivalent, or it must differ only in that some *leaves* of the pattern match *non-atomic objects* of the *macro form*. For information about error detection in this *situation*, see Section 3.5.1.7 (Destructuring Mismatch).

A destructuring *lambda list* (whether at top level or embedded) can be dotted, ending in a parameter name. This situation is treated exactly as if the parameter name that ends the *list* had appeared preceded by `&rest`.

It is permissible for a *macro form* (or a *subexpression* of a *macro form*) to be a *dotted list* only when `(... &rest var)` or `(... . var)` is used to match it. It is the responsibility of the *macro* to recognize and deal with such situations.

3.4.4.1 Destructuring by Lambda Lists

Anywhere in a *macro lambda list* where a parameter name can appear, and where *ordinary lambda list* syntax (as described in Section 3.4.1 (Ordinary Lambda Lists)) does not otherwise allow a *list*, a *destructuring lambda list* can appear in place of the parameter name. When this is done, then the argument that would match the parameter is treated as a (possibly dotted) *list*, to be used as an argument list for satisfying the parameters in the embedded *lambda list*. This is known as destructuring.

Destructuring is the process of decomposing a compound *object* into its component parts, using an abbreviated, declarative syntax, rather than writing it out by hand using the primitive component-accessing functions. Each component part is bound to a variable.

A destructuring operation requires an *object* to be decomposed, a pattern that specifies what components are to be extracted, and the names of the variables whose values are to be the components.

3.4.4.1.1 Data-directed Destructuring by Lambda Lists

In data-directed destructuring, the pattern is a sample *object* of the *type* to be decomposed. Wherever a component is to be extracted, a *symbol* appears in the pattern; this *symbol* is the name of the variable whose value will be that component.

3.4.4.1.1.1 Examples of Data-directed Destructuring by Lambda Lists

An example pattern is

```
(a b c)
```

which destructures a list of three elements. The variable `a` is assigned to the first element, `b` to the second, etc. A more complex example is

```
((first . rest) . more)
```

The important features of data-directed destructuring are its syntactic simplicity and the ability to extend it to lambda-list-directed destructuring.

3.4.4.1.2 Lambda-list-directed Destructuring by Lambda Lists

An extension of data-directed destructuring of *trees* is lambda-list-directed destructuring. This derives from the analogy between the three-element destructuring pattern

```
(first second third)
```

and the three-argument *lambda list*

```
(first second third)
```

Lambda-list-directed destructuring is identical to data-directed destructuring if no *lambda list keywords* appear in the pattern. Any list in the pattern (whether a sub-list or the whole pattern itself) that contains a *lambda list keyword* is interpreted specially. Elements of the list to the left of the first *lambda list keyword* are treated as destructuring patterns, as usual, but the remaining elements of the list are treated like a function's *lambda list* except that where a variable would normally be required, an arbitrary destructuring pattern is allowed. Note that in case of ambiguity, *lambda list* syntax is preferred over destructuring syntax. Thus, after `&optional` a list of elements is a list of a destructuring pattern and a default value form.

The detailed behavior of each *lambda list keyword* in a lambda-list-directed destructuring pattern is as follows:

`&optional`
: Each following element is a variable or a list of a destructuring pattern, a default value form, and a supplied-p variable. The default value and the supplied-p variable can be omitted. If the list being destructured ends early, so that it does not have an element to match against this destructuring (sub)-pattern, the default form is evaluated and destructured instead. The supplied-p variable receives the value **nil** if the default form is used, **t** otherwise.

`&rest`, `&body`
: The next element is a destructuring pattern that matches the rest of the list. `&body` is identical to `&rest` but declares that what is being matched is a list of forms that constitutes the body of *form*. This next element must be the last unless a *lambda list keyword* follows it.

`&aux`
: The remaining elements are not destructuring patterns at all, but are auxiliary variable bindings.

`&whole`
: The next element is a destructuring pattern that matches the entire form in a macro, or the entire *subexpression* at inner levels.

`&key`
: Each following element is one of
a *variable*,
or a list of a variable, an optional initialization form, and an optional supplied-p variable.
or a list of a list of a keyword and a destructuring pattern, an optional initialization form, and an optional supplied-p variable.
The rest of the list being destructured is taken to be alternating keywords and values and is taken apart appropriately.

`&allow-other-keys`
: Stands by itself.

3.4.5 Destructuring Lambda Lists

A *destructuring lambda list* is used by **destructuring-bind**.

Destructuring lambda lists are closely related to *macro lambda lists*; see Section 3.4.4 (Macro Lambda Lists). A *destructuring lambda list* can contain all of the *lambda list keywords* listed for *macro lambda lists* except for `&environment`, and supports destructuring in the same way. Inner *lambda lists* nested within a *macro lambda*

list have the syntax of *destructuring lambda lists*.

A *destructuring lambda list* has the following syntax:

```
reqvars::= var*

optvars::= [&optional {var | (var [init-form [supplied-p-parameter]])}*]

restvar::= [{&rest | &body} var]

keyvars::= [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}*
            [&allow-other-keys]]

auxvars::= [&aux {var | (var [init-form])}*]

envvar::= [&environment var]

wholevar::= [&whole var]

lambda-list::= (wholevar reqvars optvars restvar keyvars auxvars) |
               (wholevar reqvars optvars . var)
```

3.4.6 Boa Lambda Lists

A *boa lambda list* is a *lambda list* that is syntactically like an *ordinary lambda list*, but that is processed in "**b**y **o**rder of **a**rgument" style.

A *boa lambda list* is used only in a **defstruct** *form*, when explicitly specifying the *lambda list* of a constructor *function* (sometimes called a "boa constructor").

The `&optional`, `&rest`, `&aux`, `&key`, and `&allow-other-keys` *lambda list keywords* are recognized in a *boa lambda list*. The way these *lambda list keywords* differ from their use in an *ordinary lambda list* follows.

Consider this example, which describes how **destruct** processes its `:constructor` option.

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

This defines `create-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 *implementation-defined*. Finally, the `f` slot is initialized to contain the symbol `eff`. `&key` and `&allow-other-keys` arguments default in a manner similar to that of `&optional` arguments: if no default is supplied in the *lambda list* then the default value given in the body of the **defstruct** (if given) is used instead. For example:

```
 (defstruct (foo (:constructor CREATE-FOO (a &optional b (c 'sea)
                                             &key (d 2)
                                             &aux e (f 'eff))))
   (a 1) (b 2) (c 3) (d 4) (e 5) (f 6))

 (create-foo 10) =>  #S(FOO A 10 B 2 C SEA D 2 E implemention-dependent F EFF)
 (create-foo 10 'bee 'see :d 'dee)
=>  #S(FOO A 10 B BEE C SEE D DEE E implemention-dependent F EFF)
```

If keyword arguments of the form ((*key var*) [*default* [*svar*]]) are specified, the *slot name* is matched with *var* (not *key*).

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

If no default value is supplied for an *aux variable* variable, the consequences are undefined if an attempt is later made to read the corresponding *slot*'s value before a value is explicitly assigned. If such a *slot* has a `:type` option specified, this suppressed initialization does not imply a type mismatch situation; the declared type is only required to apply when the *slot* is finally assigned.

With this definition, the following can be written:

```
(create-foo 1 2)
```

instead of

```
(make-foo :a 1 :b 2)
```

and `create-foo` provides defaulting different from that of `make-foo`.

Additional arguments that do not correspond to slot names but are merely present to supply values used in subsequent initialization computations are allowed. For example, in the definition

```
(defstruct (frob (:constructor create-frob
                  (a &key (b 3 have-b) (c-token 'c)
                          (c (list c-token (if have-b 7 2))))))
        a b c)
```

the `c-token` argument is used merely to supply a value used in the initialization of the `c` slot. The *supplied-p parameters* associated with *optional parameters* and *keyword parameters* might also be used this way.

3.4.7 Defsetf Lambda Lists

A *defsetf lambda list* is used by **defsetf**.

A *defsetf lambda list* has the following syntax:

```
lambda-list::= (var*
                [&optional {var | (var [init-form [supplied-p-parameter]])}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
                [&environment var]
```

A *defsetf lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&environment       &optional
```

Figure 3-19. Lambda List Keywords used by Defsetf Lambda Lists

A *defsetf lambda list* differs from an *ordinary lambda list* only in that it does not permit the use of `&aux`, and that it permits use of `&environment`, which introduces an *environment parameter*.

3.4.8 Deftype Lambda Lists

A *deftype lambda list* is used by **deftype**.

A *deftype lambda list* has the same syntax as a *macro lambda list*, and can therefore contain the *lambda list keywords* as a *macro lambda list*.

A *deftype lambda list* differs from a *macro lambda list* only in that if no *init-form* is supplied for an *optional parameter* or *keyword parameter* in the *lambda-list*, the default *value* for that *parameter* is the *symbol* ***** (rather than **nil**).

3.4.9 Define-modify-macro Lambda Lists

A *define-modify-macro lambda list* is used by **define-modify-macro**.

A *define-modify-macro lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&optional  &rest
```

Figure 3-20. Lambda List Keywords used by Define-modify-macro Lambda Lists

Define-modify-macro lambda lists are similar to *ordinary lambda lists*, but do not support keyword arguments. **define-modify-macro** has no need match keyword arguments, and a *rest parameter* is sufficient. *Aux variables* are also not supported, since **define-modify-macro** has no body *forms* which could refer to such *bindings*. See the *macro* **define-modify-macro**.

3.4.10 Define-method-combination Arguments Lambda Lists

A *define-method-combination arguments lambda list* is used by the `:arguments` option to **define-method-combination**.

A *define-method-combination arguments lambda list* can contain the *lambda list keywords* shown in the next figure.

```
&allow-other-keys  &key       &rest
&aux               &optional  &whole
```

Figure 3-21. Lambda List Keywords used by Define-method-combination arguments Lambda Lists

Define-method-combination arguments lambda lists are similar to *ordinary lambda lists*, but also permit the use of `&whole`.

3.4.11 Syntactic Interaction of Documentation Strings and Declarations

In a number of situations, a *documentation string* can appear amidst a series of **declare** *expressions* prior to a series of *forms*.

In that case, if a *string* S appears where a *documentation string* is permissible and is not followed by either a **declare** *expression* or a *form* then S is taken to be a *form*; otherwise, S is taken as a *documentation string*. The consequences are unspecified if more than one such *documentation string* is present.

3.5 Error Checking in Function Calls

3.5.1 Argument Mismatch Detection

3.5.1.1 Safe and Unsafe Calls

A *call* is a *safe call* if each of the following is either *safe code* or *system code* (other than *system code* that results from *macro expansion* of *programmer code*):

* the *call*.
* the definition of the *function* being *called*.
* the point of *functional evaluation*

The following special cases require some elaboration:

* If the *function* being called is a *generic function*, it is considered *safe* if all of the following are *safe code* or *system code*:
    -- its definition (if it was defined explicitly).
    -- the *method* definitions for all *applicable methods*.
    -- the definition of its *method combination*.
* For the form `(coerce` *x* `'function)`, where *x* is a *lambda expression*, the value of the *optimize quality* **safety** in the global environment at the time the **coerce** is *executed* applies to the resulting *function*.
* For a call to the *function* **ensure-generic-function**, the value of the *optimize quality* **safety** in the *environment object* passed as the `:environment` *argument* applies to the resulting *generic function*.
* For a call to **compile** with a *lambda expression* as the *argument*, the value of the *optimize quality* **safety** in the *global environment* at the time **compile** is *called* applies to the resulting *compiled function*.
* For a call to **compile** with only one argument, if the original definition of the *function* was *safe*, then the resulting *compiled function* must also be *safe*.
* A *call* to a *method* by **call-next-method** must be considered *safe* if each of the following is *safe code* or *system code*:
    -- the definition of the *generic function* (if it was defined explicitly).
    -- the *method* definitions for all *applicable methods*.
    -- the definition of the *method combination*.
    -- the point of entry into the body of the *method defining form*, where the *binding* of **call-next-method** is established.
    -- the point of *functional evaluation* of the name **call-next-method**.

An *unsafe call* is a *call* that is not a *safe call*.

The informal intent is that the *programmer* can rely on a *call* to be *safe*, even when *system code* is involved, if all reasonable steps have been taken to ensure that the *call* is *safe*. For example, if a *programmer* calls **mapcar** from *safe code* and supplies a *function* that was *compiled* as *safe*, the *implementation* is required to ensure that **mapcar** makes a *safe call* as well.

3.5.1.1.1 Error Detection Time in Safe Calls

If an error is signaled in a *safe call*, the exact point of the *signal* is *implementation-dependent*. In particular, it might be signaled at compile time or at run time, and if signaled at run time, it might be prior to, during, or after *executing* the *call*. However, it is always prior to the execution of the body of the *function* being *called*.

3.5.1.2 Too Few Arguments

It is not permitted to supply too few *arguments* to a *function*. Too few arguments means fewer *arguments* than the number of *required parameters* for the *function*.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.3 Too Many Arguments

It is not permitted to supply too many *arguments* to a *function*. Too many arguments means more *arguments* than the number of *required parameters* plus the number of *optional parameters*; however, if the *function* uses `&rest` or `&key`, it is not possible for it to receive too many arguments.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.4 Unrecognized Keyword Arguments

It is not permitted to supply a keyword argument to a *function* using a name that is not recognized by that *function* unless keyword argument checking is suppressed as described in Section 3.4.1.4.1 (Suppressing Keyword Argument Checking).

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.5 Invalid Keyword Arguments

It is not permitted to supply a keyword argument to a *function* using a name that is not a *symbol*.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled unless keyword argument checking is suppressed as described in Section 3.4.1.4.1 (Suppressing Keyword Argument Checking); and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.6 Odd Number of Keyword Arguments

An odd number of *arguments* must not be supplied for the *keyword parameters*.

If this *situation* occurs in a *safe call*, an error of *type* **program-error** must be signaled unless keyword argument checking is suppressed as described in Section 3.4.1.4.1 (Suppressing Keyword Argument Checking); and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.7 Destructuring Mismatch

When matching a *destructuring lambda list* against a *form*, the pattern and the *form* must have compatible *tree structure*, as described in Section 3.4.4 (Macro Lambda Lists).

Otherwise, in a *safe call*, an error of *type* **program-error** must be signaled; and in an *unsafe call* the *situation* has undefined consequences.

3.5.1.8 Errors When Calling a Next Method

If **call-next-method** is called with *arguments*, the ordered set of *applicable methods* for the changed set of *arguments* for **call-next-method** must be the same as the ordered set of *applicable methods* for the original *arguments* to the *generic function*, or else an error should be signaled.

The comparison between the set of methods applicable to the new arguments and the set applicable to the original arguments is insensitive to order differences among methods with the same specializers.

If **call-next-method** is called with *arguments* that specify a different ordered set of *applicable* methods and there is no *next method* available, the test for different methods and the associated error signaling (when present) takes precedence over calling **no-next-method**.

3.6 Traversal Rules and Side Effects

The consequences are undefined when *code* executed during an *object-traversing* operation destructively modifies the *object* in a way that might affect the ongoing traversal operation. In particular, the following rules apply.

List traversal
: For *list* traversal operations, the *cdr* chain of the *list* is not allowed to be destructively modified.

Array traversal
: For *array* traversal operations, the *array* is not allowed to be adjusted and its *fill pointer*, if any, is not allowed to be changed.

Hash-table traversal
: For *hash table* traversal operations, new elements may not be added or deleted except that the element corresponding to the current hash key may be changed or removed.

Package traversal
: For *package* traversal operations (e.g., **do-symbols**), new *symbols* may not be *interned* in or *uninterned* from the *package* being traversed or any *package* that it uses except that the current *symbol* may be *uninterned* from the *package* being traversed.

3.7 Destructive Operations

3.7.1 Modification of Literal Objects

The consequences are undefined if *literal objects* are destructively modified. For this purpose, the following operations are considered *destructive*:

random-state
: Using it as an *argument* to the *function* **random**.

cons
: Changing the *car*[1] or *cdr*[1] of the *cons*, or performing a *destructive* operation on an *object* which is either the *car*[2] or the *cdr*[2] of the *cons*.

array
: Storing a new value into some element of the *array*, or performing a *destructive* operation on an *object* that is already such an *element*.

  Changing the *fill pointer*, *dimensions*, or displacement of the *array* (regardless of whether the *array* is *actually adjustable*).

Performing a *destructive* operation on another *array* that is displaced to the *array* or that otherwise shares its contents with the *array*.

hash-table
Performing a *destructive* operation on any *key*.

Storing a new *value*[4] for any *key*, or performing a *destructive* operation on any *object* that is such a *value*.

Adding or removing entries from the *hash table*.

structure-object
Storing a new value into any slot, or performing a *destructive* operation on an *object* that is the value of some slot.

standard-object
Storing a new value into any slot, or performing a *destructive* operation on an *object* that is the value of some slot.

Changing the class of the *object* (e.g., using the *function* **change-class**).

readtable
Altering the *readtable case*.

Altering the syntax type of any character in this readtable.

Altering the *reader macro function* associated with any *character* in the *readtable*, or altering the *reader macro functions* associated with *characters* defined as *dispatching macro characters* in the *readtable*.

stream
Performing I/O operations on the *stream*, or *closing* the *stream*.

All other standardized types
[This category includes, for example, **character**, **condition**, **function**, **method-combination**, **method**, **number**, **package**, **pathname**, **restart**, and **symbol**.]

There are no *standardized destructive* operations defined on *objects* of these *types*.

3.7.2 Transfer of Control during a Destructive Operation

Should a transfer of control out of a *destructive* operation occur (e.g., due to an error) the state of the *object* being modified is *implementation-dependent*.

3.7.2.1 Examples of Transfer of Control during a Destructive Operation

The following examples illustrate some of the many ways in which the *implementation-dependent* nature of the modification can manifest itself.

```
 (let ((a (list 2 1 4 3 7 6 'five)))
   (ignore-errors (sort a #'<))
   a)
=>  (1 2 3 4 6 7 FIVE)
OR=>  (2 1 4 3 7 6 FIVE)
OR=>  (2)

 (prog foo ((a (list 1 2 3 4 5 6 7 8 9 10)))
   (sort a #'(lambda (x y) (if (zerop (random 5)) (return-from foo a) (> x y)))))
=>  (1 2 3 4 5 6 7 8 9 10)
OR=>  (3 4 5 6 2 7 8 9 10 1)
OR=>  (1 2 4 3)
```