Programming Language—Common Lisp

3. Evaluation and Compilation

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

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*.

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

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*.

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

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**. Figure 3–1 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*. Figure 3–2 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.

Figure 3–3 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.

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

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**.

Figure 3–4 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
```


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

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:

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

```
(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* (here$_1$), not the inner one (here$_2$). 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 here$_2$ 4), thereby causing a return from the inner *exit point* (here$_2$). 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.

Figure 3–5 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**.

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

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*.

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

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 Figure 3–7.

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*

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

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))))
```

Figure 3–8 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*.

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

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

Figure 3–9 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

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

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

Figure 3–11 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 Figure 3–12 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 Figure 3–13.

&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

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

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)))
```


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

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 Figure 3–14.

&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 Figure 3–15 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 Figure 3–16.

&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 Figure 3–17.

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* | ↓*pattern*}*

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

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

keyvars::=[&key {*var* | ({*var* | (*keyword-name* {*var* | ↓*pattern*})} [*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*)

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

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

A *macro lambda list* can contain the *lambda list keywords* shown in Figure 3–18.

&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.

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

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* | ↓*lambda-list*}*

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

restvar::=[{`&rest` | `&body`} {*var* | ↓*lambda-list*}]

keyvars::=[`&key` {*var* | ({*var* | (*keyword-name* {*var* | ↓*lambda-list*})} [*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*)

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

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 Figure 3–19.

&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*.

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

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 Figure 3–20.

&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 Figure 3–21.

&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.

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

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.

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

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)
```


lambda *Symbol*

Syntax:

lambda *lambda-list* ⟦{*declaration*}* | *documentation*⟧ {*form*}*

Arguments:

lambda-list—an *ordinary lambda list*.

declaration—a **declare** *expression*; not evaluated.

documentation—a *string*; not evaluated.

form—a *form*.

Description:

A *lambda expression* is a *list* that can be used in place of a *function name* in certain contexts to denote a *function* by directly describing its behavior rather than indirectly by referring to the name of an *established function*.

Documentation is attached to the denoted *function* (if any is actually created) as a *documentation string*.

See Also:

function, **documentation**, Section 3.1.3 (Lambda Expressions), Section 3.1.2.1.2.4 (Lambda Forms), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

The *lambda form*

```
((lambda lambda-list . body) . arguments)
```

is semantically equivalent to the *function form*

```
(funcall #'(lambda lambda-list . body) . arguments)
```

lambda *Macro*

Syntax:

lambda *lambda-list* ⟦{*declaration*}* | *documentation*⟧ {*form*}* → *function*

Arguments and Values:

lambda-list—an *ordinary lambda list*.

declaration—a **declare** *expression*; not evaluated.

documentation—a *string*; not evaluated.

form—a *form*.

function—a *function*.

Description:

Provides a shorthand notation for a **function** *special form* involving a *lambda expression* such that:

```
   (lambda lambda-list [[{declaration}* | documentation]] {form}*)
 ≡ (function (lambda lambda-list [[{declaration}* | documentation]] {form}*))
 ≡ #'(lambda lambda-list [[{declaration}* | documentation]] {form}*)
```

Examples:

```
(funcall (lambda (x) (+ x 3)) 4) → 7
```

See Also:

lambda (symbol)

Notes:

This macro could be implemented by:

```
(defmacro lambda (&whole form &rest bvl-decls-and-body)
  (declare (ignore bvl-decls-and-body))
  `#',form)
```

compile *Function*

Syntax:

compile *name* &optional *definition* → *function, warnings-p, failure-p*

Arguments and Values:

name—a *function name*, or **nil**.

definition—a *lambda expression* or a *function*. The default is the function definition of *name* if it names a *function*, or the *macro function* of *name* if it names a *macro*. The consequences are undefined if no *definition* is supplied when the *name* is **nil**.

function—the *function-name*, or a *compiled function*.

warnings-p—a *generalized boolean*.

failure-p—a *generalized boolean*.

Description:

Compiles an *interpreted function*.

compile produces a *compiled function* from *definition*. If the *definition* is a *lambda expression*, it is coerced to a *function*. If the *definition* is already a *compiled function*, **compile** either produces that function itself (*i.e.*, is an identity operation) or an equivalent function.

If the *name* is **nil**, the resulting *compiled function* is returned directly as the *primary value*. If a *non-nil name* is given, then the resulting *compiled function* replaces the existing *function* definition of *name* and the *name* is returned as the *primary value*; if *name* is a *symbol* that names a *macro*, its *macro function* is updated and the *name* is returned as the *primary value*.

Literal objects appearing in code processed by the **compile** function are neither copied nor *coalesced*. The code resulting from the execution of **compile** references *objects* that are **eql** to the corresponding *objects* in the source code.

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

The *secondary value*, *warnings-p*, is *false* if no *conditions* of *type* **error** or **warning** were detected by the compiler, and *true* otherwise.

The *tertiary value*, *failure-p*, is *false* if no *conditions* of *type* **error** or **warning** (other than **style-warning**) were detected by the compiler, and *true* otherwise.

Examples:

```
(defun foo () "bar") → FOO
(compiled-function-p #'foo) → implementation-dependent
(compile 'foo) → FOO
(compiled-function-p #'foo) → true
(setf (symbol-function 'foo)
      (compile nil '(lambda () "replaced"))) → #<Compiled-Function>
(foo) → "replaced"
```

Affected By:

error-output, ***macroexpand-hook***.

The presence of macro definitions and proclamations.

Exceptional Situations:

The consequences are undefined if the *lexical environment* surrounding the *function* to be compiled contains any *bindings* other than those for *macros*, *symbol macros*, or *declarations*.

For information about errors detected during the compilation process, see Section 3.2.5 (Exceptional Situations in the Compiler).

See Also:

compile-file

eval *Function*

Syntax:

eval *form* → {*result*}*

Arguments and Values:

form—a *form*.

results—the *values yielded* by the *evaluation* of *form*.

Description:

Evaluates *form* in the current *dynamic environment* and the *null lexical environment*.

eval is a user interface to the evaluator.

The evaluator expands macro calls as if through the use of **macroexpand-1**.

Constants appearing in code processed by **eval** are not copied nor coalesced. The code resulting from the execution of **eval** references *objects* that are **eql** to the corresponding *objects* in the source code.

Examples:

```
(setq form '(1+ a) a 999) → 999
(eval form) → 1000
(eval 'form) → (1+ A)
(let ((a '(this would break if eval used local value))) (eval form))
→ 1000
```

See Also:

macroexpand-1, Section 3.1.2 (The Evaluation Model)

Notes:

To obtain the current dynamic value of a *symbol*, use of **symbol-value** is equivalent (and usually preferable) to use of **eval**.

Note that an **eval** *form* involves two levels of *evaluation* for its *argument*. First, *form* is *evaluated* by the normal argument evaluation mechanism as would occur with any *call*. The *object* that results from this normal *argument evaluation* becomes the *value* of the *form parameter*, and is then *evaluated* as part of the **eval** *form*. For example:

```
(eval (list 'cdr (car '((quote (a . b)) c)))) → b
```

The *argument form* `(list 'cdr (car '((quote (a . b)) c)))` is evaluated in the usual way to produce the *argument* `(cdr (quote (a . b)))`; **eval** then evaluates its *argument*, `(cdr (quote (a . b)))`, to produce `b`. Since a single *evaluation* already occurs for any *argument form* in any *function form*, **eval** is sometimes said to perform "an extra level of evaluation."

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

eval-when *Special Operator*

Syntax:

eval-when ({*situation*}*) {*form*}* → {*result*}*

Arguments and Values:

situation—One of the *symbols* `:compile-toplevel`, `:load-toplevel`, `:execute`, **compile**, **load**, or **eval**.

The use of **eval**, **compile**, and **load** is deprecated.

forms—an *implicit progn*.

results—the *values* of the *forms* if they are executed, or **nil** if they are not.

Description:

The body of an **eval-when** form is processed as an *implicit progn*, but only in the *situations* listed.

The use of the *situations* `:compile-toplevel` (or `compile`) and `:load-toplevel` (or `load`) controls whether and when *evaluation* occurs when **eval-when** appears as a *top level form* in code processed by **compile-file**. See Section 3.2.3 (File Compilation).

The use of the *situation* `:execute` (or `eval`) controls whether evaluation occurs for other **eval-when** *forms*; that is, those that are not *top level forms*, or those in code processed by **eval** or **compile**. If the `:execute` situation is specified in such a *form*, then the body *forms* are processed as an *implicit progn*; otherwise, the **eval-when** *form* returns **nil**.

eval-when normally appears as a *top level form*, but it is meaningful for it to appear as a *non-top-level form*. However, the compile-time side effects described in Section 3.2 (Compilation) only take place when **eval-when** appears as a *top level form*.

Examples:

One example of the use of **eval-when** is that for the compiler to be able to read a file properly when it uses user-defined *reader macros*, it is necessary to write

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (set-macro-character #\$ #'(lambda (stream char)
                               (declare (ignore char))
                               (list 'dollar (read stream))))) → T
```

This causes the call to **set-macro-character** to be executed in the compiler's execution environment, thereby modifying its reader syntax table.

```
;;;     The EVAL-WHEN in this case is not at toplevel, so only the :EXECUTE
;;;     keyword is considered. At compile time, this has no effect.
;;;     At load time (if the LET is at toplevel), or at execution time
;;;     (if the LET is embedded in some other form which does not execute
;;;     until later) this sets (SYMBOL-FUNCTION 'FOO1) to a function which
;;;     returns 1.
 (let ((x 1))
   (eval-when (:execute :load-toplevel :compile-toplevel)
     (setf (symbol-function 'foo1) #'(lambda () x))))

;;;     If this expression occurs at the toplevel of a file to be compiled,
;;;     it has BOTH a compile time AND a load-time effect of setting
;;;     (SYMBOL-FUNCTION 'FOO2) to a function which returns 2.
 (eval-when (:execute :load-toplevel :compile-toplevel)
   (let ((x 2))
     (eval-when (:execute :load-toplevel :compile-toplevel)
       (setf (symbol-function 'foo2) #'(lambda () x)))))

;;;     If this expression occurs at the toplevel of a file to be compiled,
;;;     it has BOTH a compile time AND a load-time effect of setting the
;;;     function cell of FOO3 to a function which returns 3.
 (eval-when (:execute :load-toplevel :compile-toplevel)
   (setf (symbol-function 'foo3) #'(lambda () 3)))

;;; #4: This always does nothing. It simply returns NIL.
 (eval-when (:compile-toplevel)
   (eval-when (:compile-toplevel)
     (print 'foo4)))

;;;     If this form occurs at toplevel of a file to be compiled, FOO5 is
;;;     printed at compile time. If this form occurs in a non-top-level
;;;     position, nothing is printed at compile time. Regardless of context,
;;;     nothing is ever printed at load time or execution time.
 (eval-when (:compile-toplevel)
   (eval-when (:execute)
     (print 'foo5)))

;;;     If this form occurs at toplevel of a file to be compiled, FOO6 is
```



```
;;;     printed at compile time.  If this form occurs in a non-top-level
;;;     position, nothing is printed at compile time. Regardless of context,
;;;     nothing is ever printed at load time or execution time.
 (eval-when (:execute :load-toplevel)
   (eval-when (:compile-toplevel)
     (print 'foo6)))
```

See Also:

compile-file, Section 3.2 (Compilation)

Notes:

The following effects are logical consequences of the definition of **eval-when**:

- Execution of a single **eval-when** expression executes the body code at most once.

- *Macros* intended for use in *top level forms* should be written so that side-effects are done by the *forms* in the macro expansion. The macro-expander itself should not do the side-effects.

  For example:

  Wrong:

  ```
  (defmacro foo ()
    (really-foo)
    `(really-foo))
  ```

  Right:

  ```
  (defmacro foo ()
    `(eval-when (:compile-toplevel :execute :load-toplevel) (really-foo)))
  ```

  Adherence to this convention means that such *macros* behave intuitively when appearing as *non-top-level forms*.

- Placing a variable binding around an **eval-when** reliably captures the binding because the compile-time-too mode cannot occur (*i.e.*, introducing a variable binding means that the **eval-when** is not a *top level form*). For example,

  ```
  (let ((x 3))
    (eval-when (:execute :load-toplevel :compile-toplevel) (print x)))
  ```

  prints 3 at execution (*i.e.*, load) time, and does not print anything at compile time. This is important so that expansions of **defun** and **defmacro** can be done in terms of **eval-when** and can correctly capture the *lexical environment*.

  ```
  (defun bar (x) (defun foo () (+ x 3)))
  ```

might expand into

```
(defun bar (x)
  (progn (eval-when (:compile-toplevel)
           (compiler::notice-function-definition 'foo '(x)))
         (eval-when (:execute :load-toplevel)
           (setf (symbol-function 'foo) #'(lambda () (+ x 3))))))
```

which would be treated by the above rules the same as

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

when the definition of `bar` is not a *top level form*.

load-time-value *Special Operator*

Syntax:

load-time-value *form* `&optional` *read-only-p* → *object*

Arguments and Values:

form—a *form*; evaluated as described below.

read-only-p—a *boolean*; not evaluated.

object—the *primary value* resulting from evaluating *form*.

Description:

load-time-value provides a mechanism for delaying evaluation of *form* until the expression is in the run-time environment; see Section 3.2 (Compilation).

Read-only-p designates whether the result can be considered a *constant object*. If **t**, the result is a read-only quantity that can, if appropriate to the *implementation*, be copied into read-only space and/or *coalesced* with *similar constant objects* from other *programs*. If **nil** (the default), the result must be neither copied nor coalesced; it must be considered to be potentially modifiable data.

If a **load-time-value** expression is processed by **compile-file**, the compiler performs its normal semantic processing (such as macro expansion and translation into machine code) on *form*, but arranges for the execution of *form* to occur at load time in a *null lexical environment*, with the result of this *evaluation* then being treated as a *literal object* at run time. It is guaranteed that the evaluation of *form* will take place only once when the *file* is *loaded*, but the order of evaluation with respect to the evaluation of *top level forms* in the file is *implementation-dependent*.

If a **load-time-value** expression appears within a function compiled with **compile**, the *form* is evaluated at compile time in a *null lexical environment*. The result of this compile-time evaluation is treated as a *literal object* in the compiled code.

If a **load-time-value** expression is processed by **eval**, *form* is evaluated in a *null lexical environment*, and one value is returned. Implementations that implicitly compile (or partially compile) expressions processed by **eval** might evaluate *form* only once, at the time this compilation is performed.

If the *same list* `(load-time-value form)` is evaluated or compiled more than once, it is *implementation-dependent* whether *form* is evaluated only once or is evaluated more than once. This can happen both when an expression being evaluated or compiled shares substructure, and when the *same form* is processed by **eval** or **compile** multiple times. Since a **load-time-value** expression can be referenced in more than one place and can be evaluated multiple times by **eval**, it is *implementation-dependent* whether each execution returns a fresh *object* or returns the same *object* as some other execution. Users must use caution when destructively modifying the resulting *object*.

If two lists `(load-time-value form)` that are the *same* under **equal** but are not *identical* are evaluated or compiled, their values always come from distinct evaluations of *form*. Their *values* may not be coalesced unless *read-only-p* is **t**.

Examples:

```
;;; The function INCR1 always returns the same value, even in different images.
;;; The function INCR2 always returns the same value in a given image,
;;; but the value it returns might vary from image to image.
(defun incr1 (x) (+ x #.(random 17)))
(defun incr2 (x) (+ x (load-time-value (random 17))))

;;; The function FOO1-REF references the nth element of the first of
;;; the *FOO-ARRAYS* that is available at load time.  It is permissible for
;;; that array to be modified (e.g., by SET-FOO1-REF); FOO1-REF will see the
;;; updated values.
(defvar *foo-arrays* (list (make-array 7) (make-array 8)))
(defun foo1-ref (n) (aref (load-time-value (first *my-arrays*) nil) n))
(defun set-foo1-ref (n val)
  (setf (aref (load-time-value (first *my-arrays*) nil) n) val))

;;; The function BAR1-REF references the nth element of the first of
;;; the *BAR-ARRAYS* that is available at load time.  The programmer has
;;; promised that the array will be treated as read-only, so the system
;;; can copy or coalesce the array.
(defvar *bar-arrays* (list (make-array 7) (make-array 8)))
(defun bar1-ref (n) (aref (load-time-value (first *my-arrays*) t) n))
```

```
;;; This use of LOAD-TIME-VALUE permits the indicated vector to be coalesced
;;; even though NIL was specified, because the object was already read-only
;;; when it was written as a literal vector rather than created by a constructor.
;;; User programs must treat the vector v as read-only.
(defun baz-ref (n)
  (let ((v (load-time-value #(A B C) nil)))
    (values (svref v n) v)))

;;; This use of LOAD-TIME-VALUE permits the indicated vector to be coalesced
;;; even though NIL was specified in the outer situation because T was specified
;;; in the inner situation.  User programs must treat the vector v as read-only.
(defun baz-ref (n)
  (let ((v (load-time-value (load-time-value (vector 1 2 3) t) nil)))
    (values (svref v n) v)))
```

See Also:

compile-file, **compile**, **eval**, Section 3.2.2.2 (Minimal Compilation), Section 3.2 (Compilation)

Notes:

load-time-value must appear outside of quoted structure in a "for *evaluation*" position. In situations which would appear to call for use of **load-time-value** within a quoted structure, the *backquote reader macro* is probably called for; see Section 2.4.6 (Backquote).

Specifying **nil** for *read-only-p* is not a way to force an object to become modifiable if it has already been made read-only. It is only a way to say that, for an object that is modifiable, this operation is not intended to make that object read-only.

quote *Special Operator*

Syntax:

quote *object* → *object*

Arguments and Values:

object—an *object*; not evaluated.

Description:

The **quote** *special operator* just returns *object*.

The consequences are undefined if *literal objects* (including *quoted objects*) are destructively modified.

Examples:

```
(setq a 1) → 1
(quote (setq a 3)) → (SETQ A 3)
a → 1
'a → A
''a → (QUOTE A)
'''a → (QUOTE (QUOTE A))
(setq a 43) → 43
(list a (cons a 3)) → (43 (43 . 3))
(list (quote a) (quote (cons a 3))) → (A (CONS A 3))
1 → 1
'1 → 1
"foo" → "foo"
'"foo" → "foo"
(car '(a b)) → A
'(car '(a b)) → (CAR (QUOTE (A B)))
#(car '(a b)) → #(CAR (QUOTE (A B)))
'#(car '(a b)) → #(CAR (QUOTE (A B)))
```

See Also:

Section 3.1 (Evaluation), Section 2.4.3 (Single-Quote), Section 3.2.1 (Compiler Terminology)

Notes:

The textual notation '*object* is equivalent to (quote *object*); see Section 3.2.1 (Compiler Terminology).

Some *objects*, called *self-evaluating objects*, do not require quotation by **quote**. However, *symbols* and *lists* are used to represent parts of programs, and so would not be useable as constant data in a program without **quote**. Since **quote** suppresses the *evaluation* of these *objects*, they become data rather than program.

compiler-macro-function *Accessor*

Syntax:

compiler-macro-function *name* &optional *environment* → *function*

(setf (compiler-macro-function *name* &optional *environment*) *new-function*)

Arguments and Values:

name—a *function name*.

environment—an *environment object*.

function, *new-function*—a *compiler macro function*, or **nil**.

Description:

Accesses the *compiler macro function* named *name*, if any, in the *environment*.

A value of **nil** denotes the absence of a *compiler macro function* named *name*.

Exceptional Situations:

The consequences are undefined if *environment* is *non-nil* in a use of **setf** of **compiler-macro-function**.

See Also:

define-compiler-macro, Section 3.2.2.1 (Compiler Macros)

define-compiler-macro *Macro*

Syntax:

define-compiler-macro *name lambda-list* ⟦ {*declaration*}* | *documentation* ⟧ {*form*}*
→ *name*

Arguments and Values:

name—a *function name*.

lambda-list—a *macro lambda list*.

declaration—a **declare** *expression*; not evaluated.

documentation—a *string*; not evaluated.

form—a *form*.

Description:

This is the normal mechanism for defining a *compiler macro function*. Its manner of definition is the same as for **defmacro**; the only differences are:

- The *name* can be a *function name* naming any *function* or *macro*.

- The expander function is installed as a *compiler macro function* for the *name*, rather than as a *macro function*.

- The **&whole** argument is bound to the form argument that is passed to the *compiler macro function*. The remaining lambda-list parameters are specified as if this form contained the function name in the *car* and the actual arguments in the *cdr*, but if the *car* of the actual form is the symbol **funcall**, then the destructuring of the arguments is actually performed using its *cddr* instead.

- *Documentation* is attached as a *documentation string* to *name* (as kind **compiler-macro**) and to the *compiler macro function*.

- Unlike an ordinary *macro*, a *compiler macro* can decline to provide an expansion merely by returning a form that is the *same* as the original (which can be obtained by using **&whole**).

Examples:

```
 (defun square (x) (expt x 2)) → SQUARE
 (define-compiler-macro square (&whole form arg)
   (if (atom arg)
       `(expt ,arg 2)
       (case (car arg)
         (square (if (= (length arg) 2)
                     `(expt ,(nth 1 arg) 4)
                     form))
         (expt   (if (= (length arg) 3)
                     (if (numberp (nth 2 arg))
                         `(expt ,(nth 1 arg) ,(* 2 (nth 2 arg)))
                         `(expt ,(nth 1 arg) (* 2 ,(nth 2 arg))))
                     form))
         (otherwise `(expt ,arg 2))))) → SQUARE
 (square (square 3)) → 81
 (macroexpand '(square x)) → (SQUARE X), false
 (funcall (compiler-macro-function 'square) '(square x) nil)
→ (EXPT X 2)
 (funcall (compiler-macro-function 'square) '(square (square x)) nil)
→ (EXPT X 4)
 (funcall (compiler-macro-function 'square) '(funcall #'square x) nil)
→ (EXPT X 2)

 (defun distance-positional (x1 y1 x2 y2)
   (sqrt (+ (expt (- x2 x1) 2) (expt (- y2 y1) 2))))
→ DISTANCE-POSITIONAL
 (defun distance (&key (x1 0) (y1 0) (x2 x1) (y2 y1))
   (distance-positional x1 y1 x2 y2))
→ DISTANCE
 (define-compiler-macro distance (&whole form
                                  &rest key-value-pairs
```

```
                                        &key (x1 0  x1-p)
                                             (y1 0  y1-p)
                                             (x2 x1 x2-p)
                                             (y2 y1 y2-p)
                                        &allow-other-keys
                                        &environment env)
          (flet ((key (n) (nth (* n 2) key-value-pairs))
                 (arg (n) (nth (1+ (* n 2)) key-value-pairs))
                 (simplep (x)
                   (let ((expanded-x (macroexpand x env)))
                     (or (constantp expanded-x env)
                         (symbolp expanded-x)))))
            (let ((n (/ (length key-value-pairs) 2)))
              (multiple-value-bind (x1s y1s x2s y2s others)
                  (loop for (key) on key-value-pairs by #'cddr
                        count (eq key ':x1) into x1s
                        count (eq key ':y1) into y1s
                        count (eq key ':x2) into x2s
                        count (eq key ':y1) into y2s
                        count (not (member key '(:x1 :x2 :y1 :y2)))
                          into others
                        finally (return (values x1s y1s x2s y2s others)))
                (cond ((and (= n 4)
                            (eq (key 0) :x1)
                            (eq (key 1) :y1)
                            (eq (key 2) :x2)
                            (eq (key 3) :y2))
                       `(distance-positional ,x1 ,y1 ,x2 ,y2))
                      ((and (if x1-p (and (= x1s 1) (simplep x1)) t)
                            (if y1-p (and (= y1s 1) (simplep y1)) t)
                            (if x2-p (and (= x2s 1) (simplep x2)) t)
                            (if y2-p (and (= y2s 1) (simplep y2)) t)
                            (zerop others))
                       `(distance-positional ,x1 ,y1 ,x2 ,y2))
                      ((and (< x1s 2) (< y1s 2) (< x2s 2) (< y2s 2)
                            (zerop others))
                       (let ((temps (loop repeat n collect (gensym))))
                         `(let ,(loop for i below n
                                      collect (list (nth i temps) (arg i)))
                            (distance
                              ,@(loop for i below n
                                      append (list (key i) (nth i temps)))))))
                      (t form))))))
→ DISTANCE
 (dolist (form
```



```
                 '((distance :x1 (setq x 7) :x2 (decf x) :y1 (decf x) :y2 (decf x))
                   (distance :x1 (setq x 7) :y1 (decf x) :x2 (decf x) :y2 (decf x))
                   (distance :x1 (setq x 7) :y1 (incf x))
                   (distance :x1 (setq x 7) :y1 (incf x) :x1 (incf x))
                   (distance :x1 a1 :y1 b1 :x2 a2 :y2 b2)
                   (distance :x1 a1 :x2 a2 :y1 b1 :y2 b2)
                   (distance :x1 a1 :y1 b1 :z1 c1 :x2 a2 :y2 b2 :z2 c2)))
   (print (funcall (compiler-macro-function 'distance) form nil)))
▷ (LET ((#:G6558 (SETQ X 7))
▷       (#:G6559 (DECF X))
▷       (#:G6560 (DECF X))
▷       (#:G6561 (DECF X)))
▷   (DISTANCE :X1 #:G6558 :X2 #:G6559 :Y1 #:G6560 :Y2 #:G6561))
▷ (DISTANCE-POSITIONAL (SETQ X 7) (DECF X) (DECF X) (DECF X))
▷ (LET ((#:G6567 (SETQ X 7))
▷       (#:G6568 (INCF X)))
▷   (DISTANCE :X1 #:G6567 :Y1 #:G6568))
▷ (DISTANCE :X1 (SETQ X 7) :Y1 (INCF X) :X1 (INCF X))
▷ (DISTANCE-POSITIONAL A1 B1 A2 B2)
▷ (DISTANCE-POSITIONAL A1 B1 A2 B2)
▷ (DISTANCE :X1 A1 :Y1 B1 :Z1 C1 :X2 A2 :Y2 B2 :Z2 C2)
→ NIL
```

See Also:

compiler-macro-function, **defmacro**, **documentation**, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

The consequences of writing a *compiler macro* definition for a function in the `COMMON-LISP` *package* are undefined; it is quite possible that in some *implementations* such an attempt would override an equivalent or equally important definition. In general, it is recommended that a programmer only write *compiler macro* definitions for *functions* he or she personally maintains–writing a *compiler macro* definition for a function maintained elsewhere is normally considered a violation of traditional rules of modularity and data abstraction.

defmacro *Macro*

Syntax:

defmacro *name lambda-list* ⟦ {*declaration*}* | *documentation* ⟧ {*form*}*
  → *name*

Arguments and Values:

name—a *symbol*.

lambda-list—a *macro lambda list*.

declaration—a **declare** *expression*; not evaluated.

documentation—a *string*; not evaluated.

form—a *form*.

Description:

Defines *name* as a *macro* by associating a *macro function* with that *name* in the global environment. The *macro function* is defined in the same *lexical environment* in which the **defmacro** *form* appears.

The parameter variables in *lambda-list* are bound to destructured portions of the macro call.

The expansion function accepts two arguments, a *form* and an *environment*. The expansion function returns a *form*. The body of the expansion function is specified by *forms*. *Forms* are executed in order. The value of the last *form* executed is returned as the expansion of the *macro*. The body *forms* of the expansion function (but not the *lambda-list*) are implicitly enclosed in a *block* whose name is *name*.

The *lambda-list* conforms to the requirements described in Section 3.4.4 (Macro Lambda Lists).

Documentation is attached as a *documentation string* to *name* (as kind **function**) and to the *macro function*.

defmacro can be used to redefine a *macro* or to replace a *function* definition with a *macro* definition.

Recursive expansion of the *form* returned must terminate, including the expansion of other *macros* which are *subforms* of other *forms* returned.

The consequences are undefined if the result of fully macroexpanding a *form* contains any *circular list structure* except in *literal objects*.

If a **defmacro** *form* appears as a *top level form*, the *compiler* must store the *macro* definition at compile time, so that occurrences of the macro later on in the file can be expanded correctly. Users must ensure that the body of the *macro* can be evaluated at compile time if it is referenced within the *file* being *compiled*.

Examples:

```
(defmacro mac1 (a b) "Mac1 multiplies and adds"
         `(+ ,a (* ,b 3))) → MAC1
(mac1 4 5) → 19
(documentation 'mac1 'function) → "Mac1 multiplies and adds"
(defmacro mac2 (&optional (a 2 b) (c 3 d) &rest x) `'(,a ,b ,c ,d ,x)) → MAC2
(mac2 6) → (6 T 3 NIL NIL)
```


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

defmacro

```
(mac2 6 3 8) → (6 T 3 T (8))
(defmacro mac3 (&whole r a &optional (b 3) &rest x &key c (d a))
   `'(,r ,a ,b ,c ,d ,x)) → MAC3
(mac3 1 6 :d 8 :c 9 :d 10) → ((MAC3 1 6 :D 8 :C 9 :D 10) 1 6 9 8 (:D 8 :C 9 :D 10))
```

The stipulation that an embedded *destructuring lambda list* is permitted only where *ordinary lambda list* syntax would permit a parameter name but not a *list* is made to prevent ambiguity. For example, the following is not valid:

```
(defmacro loser (x &optional (a b &rest c) &rest z)
  ...)
```

because *ordinary lambda list* syntax does permit a *list* following `&optional`; the list `(a b &rest c)` would be interpreted as describing an optional parameter named `a` whose default value is that of the form `b`, with a supplied-p parameter named **&rest** (not valid), and an extraneous symbol `c` in the list (also not valid). An almost correct way to express this is

```
(defmacro loser (x &optional ((a b &rest c)) &rest z)
  ...)
```

The extra set of parentheses removes the ambiguity. However, the definition is now incorrect because a macro call such as `(loser (car pool))` would not provide any argument form for the lambda list `(a b &rest c)`, and so the default value against which to match the *lambda list* would be **nil** because no explicit default value was specified. The consequences of this are unspecified since the empty list, **nil**, does not have *forms* to satisfy the parameters `a` and `b`. The fully correct definition would be either

```
(defmacro loser (x &optional ((a b &rest c) '(nil nil)) &rest z)
  ...)
```

or

```
(defmacro loser (x &optional ((&optional a b &rest c)) &rest z)
  ...)
```

These differ slightly: the first requires that if the macro call specifies `a` explicitly then it must also specify `b` explicitly, whereas the second does not have this requirement. For example,

```
(loser (car pool) ((+ x 1)))
```

would be a valid call for the second definition but not for the first.

```
(defmacro dm1a (&whole x) `',x)
(macroexpand '(dm1a))  → (QUOTE (DM1A))
(macroexpand '(dm1a a)) is an error.

(defmacro dm1b (&whole x a &optional b) `'(,x ,a ,b))
(macroexpand '(dm1b))  is an error.
(macroexpand '(dm1b q))  → (QUOTE ((DM1B Q) Q NIL))
```

```
(macroexpand '(dm1b q r)) → (QUOTE ((DM1B Q R) Q R))
(macroexpand '(dm1b q r s)) is an error.

(defmacro dm2a (&whole form a b) `'(form ,form a ,a b ,b))
(macroexpand '(dm2a x y)) → (QUOTE (FORM (DM2A X Y) A X B Y))
(dm2a x y) → (FORM (DM2A X Y) A X B Y)

(defmacro dm2b (&whole form a (&whole b (c . d) &optional (e 5))
                &body f &environment env)
  ``(,',form ,,a ,',b ,',(macroexpand c env) ,',d ,',e ,',f))
;Note that because backquote is involved, implementations may differ
;slightly in the nature (though not the functionality) of the expansion.
(macroexpand '(dm2b x1 (((incf x2) x3 x4)) x5 x6))
→ (LIST* '(DM2B X1 (((INCF X2) X3 X4))
                X5 X6)
          X1
          '((((INCF X2) X3 X4)) (SETQ X2 (+ X2 1)) (X3 X4) 5 (X5 X6))),
   T
(let ((x1 5))
  (macrolet ((segundo (x) `(cadr ,x)))
    (dm2b x1 (((segundo x2) x3 x4)) x5 x6)))
→ ((DM2B X1 (((SEGUNDO X2) X3 X4)) X5 X6)
     5 (((SEGUNDO X2) X3 X4)) (CADR X2) (X3 X4) 5 (X5 X6))
```

See Also:

define-compiler-macro, **destructuring-bind**, **documentation**, **macroexpand**, ***macroexpand-hook***, **macrolet**, **macro-function**, Section 3.1 (Evaluation), Section 3.2 (Compilation), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

macro-function *Accessor*

Syntax:

macro-function *symbol* `&optional` *environment* → *function*

(**setf** (**macro-function** *symbol* `&optional` *environment*) *new-function*)

Arguments and Values:

symbol—a *symbol*.

environment—an *environment object*.

function—a *macro function* or **nil**.

new-function—a *macro function*.

Description:

Determines whether *symbol* has a function definition as a macro in the specified *environment*.

If so, the macro expansion function, a function of two arguments, is returned. If *symbol* has no function definition in the lexical environment *environment*, or its definition is not a *macro*, **macro-function** returns **nil**.

It is possible for both **macro-function** and **special-operator-p** to return *true* of *symbol*. The *macro* definition must be available for use by programs that understand only the standard Common Lisp *special forms*.

Examples:

```
(defmacro macfun (x) '(macro-function 'macfun)) → MACFUN
(not (macro-function 'macfun)) → false


(macrolet ((foo (&environment env)
              (if (macro-function 'bar env)
                 ''yes
                 ''no)))
   (list (foo)
         (macrolet ((bar () :beep))
            (foo))))

→ (NO YES)
```

Affected By:

`(setf macro-function)`, **defmacro**, and **macrolet**.

Exceptional Situations:

The consequences are undefined if *environment* is *non-nil* in a use of **setf** of **macro-function**.

See Also:

defmacro, Section 3.1 (Evaluation)

Notes:

setf can be used with **macro-function** to install a *macro* as a symbol's global function definition:

```
(setf (macro-function symbol) fn)
```

The value installed must be a *function* that accepts two arguments, the entire macro call and an *environment*, and computes the expansion for that call. Performing this operation causes *symbol* to have only that macro definition as its global function definition; any previous definition, whether as a *macro* or as a *function*, is lost.

macroexpand, macroexpand-1 *Function*

Syntax:

macroexpand *form* &optional *env* → *expansion, expanded-p*

macroexpand-1 *form* &optional *env* → *expansion, expanded-p*

Arguments and Values:

form—a *form*.

env—an *environment object*. The default is **nil**.

expansion—a *form*.

expanded-p—a *generalized boolean*.

Description:

macroexpand and **macroexpand-1** expand *macros*.

If *form* is a *macro form*, then **macroexpand-1** expands the *macro form* call once.

macroexpand repeatedly expands *form* until it is no longer a *macro form*. In effect, **macroexpand** calls **macroexpand-1** repeatedly until the *secondary value* it returns is **nil**.

If *form* is a *macro form*, then the *expansion* is a *macro expansion* and *expanded-p* is *true*. Otherwise, the *expansion* is the given *form* and *expanded-p* is *false*.

Macro expansion is carried out as follows. Once **macroexpand-1** has determined that the *form* is a *macro form*, it obtains an appropriate expansion *function* for the *macro* or *symbol macro*. The value of ***macroexpand-hook*** is coerced to a *function* and then called as a *function* of three arguments: the expansion *function*, the *form*, and the *env*. The *value* returned from this call is taken to be the expansion of the *form*.

In addition to *macro* definitions in the global environment, any local macro definitions established within *env* by **macrolet** or **symbol-macrolet** are considered. If only *form* is supplied as an argument, then the environment is effectively null, and only global macro definitions as established by **defmacro** are considered. *Macro* definitions are shadowed by local *function* definitions.

Examples:

```
 (defmacro alpha (x y) `(beta ,x ,y)) → ALPHA
```



```
 (defmacro beta (x y) `(gamma ,x ,y)) → BETA
 (defmacro delta (x y) `(gamma ,x ,y)) → EPSILON
 (defmacro expand (form &environment env)
   (multiple-value-bind (expansion expanded-p)
       (macroexpand form env)
     `(values ',expansion ',expanded-p))) → EXPAND
 (defmacro expand-1 (form &environment env)
   (multiple-value-bind (expansion expanded-p)
       (macroexpand-1 form env)
     `(values ',expansion ',expanded-p))) → EXPAND-1


;; Simple examples involving just the global environment
 (macroexpand-1 '(alpha a b)) → (BETA A B), true
 (expand-1 (alpha a b)) → (BETA A B), true
 (macroexpand '(alpha a b)) → (GAMMA A B), true
 (expand (alpha a b)) → (GAMMA A B), true
 (macroexpand-1 'not-a-macro) → NOT-A-MACRO, false
 (expand-1 not-a-macro) → NOT-A-MACRO, false
 (macroexpand '(not-a-macro a b)) → (NOT-A-MACRO A B), false
 (expand (not-a-macro a b)) → (NOT-A-MACRO A B), false


;; Examples involving lexical environments
 (macrolet ((alpha (x y) `(delta ,x ,y)))
   (macroexpand-1 '(alpha a b))) → (BETA A B), true
 (macrolet ((alpha (x y) `(delta ,x ,y)))
   (expand-1 (alpha a b))) → (DELTA A B), true
 (macrolet ((alpha (x y) `(delta ,x ,y)))
   (macroexpand '(alpha a b))) → (GAMMA A B), true
 (macrolet ((alpha (x y) `(delta ,x ,y)))
   (expand (alpha a b))) → (GAMMA A B), true
 (macrolet ((beta (x y) `(epsilon ,x ,y)))
   (expand (alpha a b))) → (EPSILON A B), true
 (let ((x (list 1 2 3)))
   (symbol-macrolet ((a (first x)))
     (expand a))) → (FIRST X), true
 (let ((x (list 1 2 3)))
   (symbol-macrolet ((a (first x)))
     (macroexpand 'a))) → A, false
 (symbol-macrolet ((b (alpha x y)))
   (expand-1 b)) → (ALPHA X Y), true
 (symbol-macrolet ((b (alpha x y)))
   (expand b)) → (GAMMA X Y), true
 (symbol-macrolet ((b (alpha x y))
```

```
                   (a b))
   (expand-1 a)) → B, true
 (symbol-macrolet ((b (alpha x y))
                   (a b))
   (expand a)) → (GAMMA X Y), true


;; Examples of shadowing behavior
 (flet ((beta (x y) (+ x y)))
   (expand (alpha a b))) → (BETA A B), true
 (macrolet ((alpha (x y) `(delta ,x ,y)))
   (flet ((alpha (x y) (+ x y)))
     (expand (alpha a b)))) → (ALPHA A B), false
 (let ((x (list 1 2 3)))
   (symbol-macrolet ((a (first x)))
     (let ((a x))
       (expand a)))) → A, false
```

Affected By:

defmacro, **setf** of **macro-function**, **macrolet**, **symbol-macrolet**

See Also:

macroexpand-hook, **defmacro**, **setf** of **macro-function**, **macrolet**, **symbol-macrolet**, Section 3.1 (Evaluation)

Notes:

Neither **macroexpand** nor **macroexpand-1** makes any explicit attempt to expand *macro forms* that are either *subforms* of the *form* or *subforms* of the *expansion*. Such expansion might occur implicitly, however, due to the semantics or implementation of the *macro function*.

define-symbol-macro *Macro*

Syntax:

define-symbol-macro *symbol expansion*
→ *symbol*

Arguments and Values:

symbol—a *symbol*.

expansion—a *form*.

define-symbol-macro

Description:

Provides a mechanism for globally affecting the *macro expansion* of the indicated *symbol*.

Globally establishes an expansion function for the *symbol macro* named by *symbol*. The only guaranteed property of an expansion *function* for a *symbol macro* is that when it is applied to the *form* and the *environment* it returns the correct expansion. (In particular, it is *implementation-dependent* whether the expansion is conceptually stored in the expansion function, the *environment*, or both.)

Each global reference to *symbol* (*i.e.*, not *shadowed*$_2$ by a *binding* for a *variable* or *symbol macro* named by the same *symbol*) is expanded by the normal macro expansion process; see Section 3.1.2.1.1 (Symbols as Forms). The expansion of a *symbol macro* is subject to further *macro expansion* in the same *lexical environment* as the *symbol macro* reference, exactly analogous to normal *macros*.

The consequences are unspecified if a **special** declaration is made for *symbol* while in the scope of this definition (*i.e.*, when it is not *shadowed*$_2$ by a *binding* for a *variable* or *symbol macro* named by the same *symbol*).

Any use of **setq** to set the value of the *symbol* while in the scope of this definition is treated as if it were a **setf**. **psetq** of *symbol* is treated as if it were a **psetf**, and **multiple-value-setq** is treated as if it were a **setf** of **values**.

A *binding* for a *symbol macro* can be *shadowed*$_2$ by **let** or **symbol-macrolet**.

Examples:

```
(defvar *things* (list 'alpha 'beta 'gamma)) → *THINGS*

(define-symbol-macro thing1 (first *things*)) → THING1
(define-symbol-macro thing2 (second *things*)) → THING2
(define-symbol-macro thing3 (third *things*)) → THING3

thing1 → ALPHA
(setq thing1 'ONE) → ONE
*things* → (ONE BETA GAMMA)
(multiple-value-setq (thing2 thing3) (values 'two 'three)) → TWO
thing3 → THREE
*things* → (ONE TWO THREE)

(list thing2 (let ((thing2 2)) thing2)) → (TWO 2)
```

Exceptional Situations:

If *symbol* is already defined as a *global variable*, an error of *type* **program-error** is signaled.

See Also:

symbol-macrolet, **macroexpand**

symbol-macrolet *Special Operator*

Syntax:

symbol-macrolet ({(*symbol expansion*)}*) {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

symbol—a *symbol*.

expansion—a *form*.

declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

results—the *values* returned by the *forms*.

Description:

symbol-macrolet provides a mechanism for affecting the *macro expansion* environment for *symbols*.

symbol-macrolet lexically establishes expansion functions for each of the *symbol macros* named by *symbols*. The only guaranteed property of an expansion *function* for a *symbol macro* is that when it is applied to the *form* and the *environment* it returns the correct expansion. (In particular, it is *implementation-dependent* whether the expansion is conceptually stored in the expansion function, the *environment*, or both.)

Each reference to *symbol* as a variable within the lexical *scope* of **symbol-macrolet** is expanded by the normal macro expansion process; see Section 3.1.2.1.1 (Symbols as Forms). The expansion of a symbol macro is subject to further macro expansion in the same lexical environment as the symbol macro invocation, exactly analogous to normal *macros*.

Exactly the same *declarations* are allowed as for **let** with one exception: **symbol-macrolet** signals an error if a **special** declaration names one of the *symbols* being defined by **symbol-macrolet**.

When the *forms* of the **symbol-macrolet** form are expanded, any use of **setq** to set the value of one of the specified variables is treated as if it were a **setf**. **psetq** of a *symbol* defined as a symbol macro is treated as if it were a **psetf**, and **multiple-value-setq** is treated as if it were a **setf** of **values**.

The use of **symbol-macrolet** can be shadowed by **let**. In other words, **symbol-macrolet** only substitutes for occurrences of *symbol* that would be in the *scope* of a lexical binding of *symbol* surrounding the *forms*.

Examples:

```
;;; The following is equivalent to
;;;   (list 'foo (let ((x 'bar)) x)),
;;; not
;;;   (list 'foo (let (('foo 'bar)) 'foo))
 (symbol-macrolet ((x 'foo))
   (list x (let ((x 'bar)) x)))
→ (foo bar)
not→ (foo foo)

 (symbol-macrolet ((x '(foo x)))
   (list x))
→ ((FOO X))
```

Exceptional Situations:

If an attempt is made to bind a *symbol* that is defined as a *global variable*, an error of *type* **program-error** is signaled.

If *declaration* contains a **special** declaration that names one of the *symbols* being bound by **symbol-macrolet**, an error of *type* **program-error** is signaled.

See Also:

with-slots, **macroexpand**

Notes:

The special form **symbol-macrolet** is the basic mechanism that is used to implement **with-slots**.

If a **symbol-macrolet** *form* is a *top level form*, the *forms* are also processed as *top level forms*. See Section 3.2.3 (File Compilation).

macroexpand-hook *Variable*

Value Type:

a *designator* for a *function* of three *arguments*: a *macro function*, a *macro form*, and an *environment object*.

Initial Value:

a *designator* for a function that is equivalent to the *function* **funcall**, but that might have additional *implementation-dependent* side-effects.

Description:

Used as the expansion interface hook by **macroexpand-1** to control the *macro expansion* process. When a *macro form* is to be expanded, this *function* is called with three arguments: the *macro function*, the *macro form*, and the *environment* in which the *macro form* is to be expanded. The *environment object* has *dynamic extent*; the consequences are undefined if the *environment object* is referred to outside the *dynamic extent* of the macro expansion function.

Examples:

```
(defun hook (expander form env)
   (format t "Now expanding: ~S~%" form)
   (funcall expander form env)) → HOOK
(defmacro machook (x y) `(/ (+ ,x ,y) 2)) → MACHOOK
(macroexpand '(machook 1 2)) → (/ (+ 1 2) 2), true
(let ((*macroexpand-hook* #'hook)) (macroexpand '(machook 1 2)))
▷ Now expanding (MACHOOK 1 2)
→ (/ (+ 1 2) 2), true
```

See Also:

macroexpand, **macroexpand-1**, **funcall**, Section 3.1 (Evaluation)

Notes:

The net effect of the chosen initial value is to just invoke the *macro function*, giving it the *macro form* and *environment* as its two arguments.

Users or user programs can *assign* this *variable* to customize or trace the *macro expansion* mechanism. Note, however, that this *variable* is a global resource, potentially shared by multiple *programs*; as such, if any two *programs* depend for their correctness on the setting of this *variable*, those *programs* may not be able to run in the same *Lisp image*. For this reason, it is frequently best to confine its uses to debugging situations.

Users who put their own function into ***macroexpand-hook*** should consider saving the previous value of the hook, and calling that value from their own.

proclaim *Function*

Syntax:

proclaim *declaration-specifier* → *implementation-dependent*

Arguments and Values:

declaration-specifier—a *declaration specifier*.

Description:

Establishes the *declaration* specified by *declaration-specifier* in the *global environment*.

Such a *declaration*, sometimes called a *global declaration* or a *proclamation*, is always in force unless locally *shadowed*.

Names of *variables* and *functions* within *declaration-specifier* refer to *dynamic variables* and global *function* definitions, respectively.

Figure 3–22 shows a list of *declaration identifiers* that can be used with **proclaim**.

declaration	**inline**	**optimize**	**type**
ftype	**notinline**	**special**	

Figure 3–22. Global Declaration Specifiers

An implementation is free to support other (*implementation-defined*) *declaration identifiers* as well.

Examples:

```
(defun declare-variable-types-globally (type vars)
  (proclaim `(type ,type ,@vars))
  type)

;; Once this form is executed, the dynamic variable *TOLERANCE*
;; must always contain a float.
(declare-variable-types-globally 'float '(*tolerance*))
→ FLOAT
```

See Also:

declaim, **declare**, Section 3.2 (Compilation)

Notes:

Although the *execution* of a **proclaim** *form* has effects that might affect compilation, the compiler does not make any attempt to recognize and specially process **proclaim** *forms*. A *proclamation* such as the following, even if a *top level form*, does not have any effect until it is executed:

```
(proclaim '(special *x*))
```

If compile time side effects are desired, **eval-when** may be useful. For example:

```
(eval-when (:execute :compile-toplevel :load-toplevel)
  (proclaim '(special *x*)))
```

In most such cases, however, it is preferrable to use **declaim** for this purpose.

Since **proclaim** *forms* are ordinary *function forms*, *macro forms* can expand into them.

declaim *Macro*

Syntax:

declaim {*declaration-specifier*}* → *implementation-dependent*

Arguments and Values:

declaration-specifier—a *declaration specifier*; not evaluated.

Description:

Establishes the *declarations* specified by the *declaration-specifiers*.

If a use of this macro appears as a *top level form* in a *file* being processed by the *file compiler*, the proclamations are also made at compile-time. As with other defining macros, it is unspecified whether or not the compile-time side-effects of a **declaim** persist after the *file* has been *compiled*.

Examples:

See Also:

declare, **proclaim**

declare *Symbol*

Syntax:

declare {*declaration-specifier*}*

Arguments:

declaration-specifier—a *declaration specifier*; not evaluated.

Description:

A **declare** *expression*, sometimes called a *declaration*, can occur only at the beginning of the bodies of certain *forms*; that is, it may be preceded only by other **declare** *expressions*, or by a *documentation string* if the context permits.

A **declare** *expression* can occur in a *lambda expression* or in any of the *forms* listed in Figure 3–23.

defgeneric	**do-external-symbols**	**prog**
define-compiler-macro	**do-symbols**	**prog***
define-method-combination	**dolist**	**restart-case**
define-setf-expander	**dotimes**	**symbol-macrolet**
defmacro	**flet**	**with-accessors**
defmethod	**handler-case**	**with-hash-table-iterator**
defsetf	**labels**	**with-input-from-string**
deftype	**let**	**with-open-file**
defun	**let***	**with-open-stream**
destructuring-bind	**locally**	**with-output-to-string**
do	**macrolet**	**with-package-iterator**
do*	**multiple-value-bind**	**with-slots**
do-all-symbols	**pprint-logical-block**	

Figure 3–23. Standardized Forms In Which Declarations Can Occur

A **declare** *expression* can only occur where specified by the syntax of these *forms*. The consequences of attempting to evaluate a **declare** *expression* are undefined. In situations where such *expressions* can appear, explicit checks are made for their presence and they are never actually evaluated; it is for this reason that they are called "**declare** *expressions*" rather than "**declare** *forms*."

Macro forms cannot expand into declarations; **declare** *expressions* must appear as actual *subexpressions* of the *form* to which they refer.

Figure 3–24 shows a list of *declaration identifiers* that can be used with **declare**.

dynamic-extent	**ignore**	**optimize**
ftype	**inline**	**special**
ignorable	**notinline**	**type**

Figure 3–24. Local Declaration Specifiers

An implementation is free to support other (*implementation-defined*) *declaration identifiers* as well.

Examples:

```
(defun nonsense (k x z)
  (foo z x)                     ;First call to foo
  (let ((j (foo k x))           ;Second call to foo
        (x (* k k)))
    (declare (inline foo) (special x z))
    (foo x j z)))               ;Third call to foo
```

In this example, the **inline** declaration applies only to the third call to `foo`, but not to the first or second ones. The **special** declaration of `x` causes **let** to make a dynamic *binding* for `x`, and causes the reference to `x` in the body of **let** to be a dynamic reference. The reference to `x` in the second call to `foo` is a local reference to the second parameter of `nonsense`. The reference to `x` in the first call to `foo` is a local reference, not a **special** one. The **special** declaration of `z` causes the reference to `z` in the third call to `foo` to be a dynamic reference; it does not refer to the parameter to `nonsense` named `z`, because that parameter *binding* has not been declared to be **special**. (The **special** declaration of `z` does not appear in the body of **defun**, but in an inner *form*, and therefore does not affect the *binding* of the *parameter*.)

Exceptional Situations:

The consequences of trying to use a **declare** *expression* as a *form* to be *evaluated* are undefined.

See Also:

proclaim, Section 4.2.3 (Type Specifiers), **declaration**, **dynamic-extent**, **ftype**, **ignorable**, **ignore**, **inline**, **notinline**, **optimize**, **type**

ignore, ignorable *Declaration*

Syntax:

(ignore {*var* | (**function** *fn*)}*)

(ignorable {*var* | (**function** *fn*)}*)

Arguments:

var—a *variable name*.

fn—a *function name*.

Valid Context:

declaration

Binding Types Affected:

variable, *function*

Description:

The **ignore** and **ignorable** declarations refer to *for-value references* to *variable bindings* for the *vars* and to *function bindings* for the *fns*.

An **ignore** *declaration* specifies that *for-value references* to the indicated *bindings* will not occur within the scope of the *declaration*. Within the *scope* of such a *declaration*, it is desirable for a compiler to issue a warning about the presence of either a *for-value reference* to any *var* or *fn*, or a **special** *declaration* for any *var*.

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

An **ignorable** *declaration* specifies that *for-value references* to the indicated *bindings* might or might not occur within the scope of the *declaration*. Within the *scope* of such a *declaration*, it is not desirable for a compiler to issue a warning about the presence or absence of either a *for-value reference* to any *var* or *fn*, or a **special** *declaration* for any *var*.

When not within the *scope* of a **ignore** or **ignorable** *declaration*, it is desirable for a compiler to issue a warning about any *var* for which there is neither a *for-value reference* nor a **special** *declaration*, or about any *fn* for which there is no *for-value reference*.

Any warning about a "used" or "unused" *binding* must be of *type* **style-warning**, and may not affect program semantics.

The *stream variables* established by **with-open-file**, **with-open-stream**, **with-input-from-string**, and **with-output-to-string**, and all *iteration variables* are, by definition, always "used". Using `(declare (ignore v))`, for such a *variable* *v* has unspecified consequences.

See Also:

declare

dynamic-extent *Declaration*

Syntax:

(dynamic-extent ⟦ {*var*}* | (**function** *fn*)* ⟧)

Arguments:

var—a *variable name*.

fn—a *function name*.

Valid Context:

declaration

Binding Types Affected:

variable, *function*

Description:

In some containing *form*, F, this declaration asserts for each var_i (which need not be bound by F), and for each *value* v_{ij} that var_i takes on, and for each *object* x_{ijk} that is an *otherwise inaccessible part* of v_{ij} at any time when v_{ij} becomes the value of var_i, that just after the execution of F terminates, x_{ijk} is either *inaccessible* (if F established a *binding* for var_i) or still an *otherwise inaccessible part* of the current value of var_i (if F did not establish a *binding* for var_i). The same relation holds for each fn_i, except that the *bindings* are in the *function namespace*.

The compiler is permitted to use this information in any way that is appropriate to the *implementation* and that does not conflict with the semantics of Common Lisp.

dynamic-extent declarations can be *free declarations* or *bound declarations*.

The *vars* and *fns* named in a **dynamic-extent** declaration must not refer to *symbol macro* or *macro* bindings.

Examples:

Since stack allocation of the initial value entails knowing at the *object*'s creation time that the *object* can be *stack-allocated*, it is not generally useful to make a **dynamic-extent** *declaration* for *variables* which have no lexically apparent initial value. For example, it is probably useful to write:

```
(defun f ()
  (let ((x (list 1 2 3)))
    (declare (dynamic-extent x))
        ...))
```

This would permit those compilers that wish to do so to *stack allocate* the list held by the local variable x. It is permissible, but in practice probably not as useful, to write:

```
(defun g (x) (declare (dynamic-extent x)) ...)
(defun f () (g (list 1 2 3)))
```

Most compilers would probably not *stack allocate* the *argument* to g in f because it would be a modularity violation for the compiler to assume facts about g from within f. Only an implementation that was willing to be responsible for recompiling f if the definition of g changed incompatibly could legitimately *stack allocate* the *list* argument to g in f.

Here is another example:

```
(declaim (inline g))
(defun g (x) (declare (dynamic-extent x)) ...)
(defun f () (g (list 1 2 3)))

(defun f ()
  (flet ((g (x) (declare (dynamic-extent x)) ...))
    (g (list 1 2 3))))
```

In the previous example, some compilers might determine that optimization was possible and others might not.

A variant of this is the so-called "stack allocated rest list" that can be achieved (in implementations supporting the optimization) by:

```
(defun f (&rest x)
  (declare (dynamic-extent x))
  ...)
```

Note that although the initial value of x is not explicit, the f function is responsible for assembling the list x from the passed arguments, so the f function can be optimized by the compiler to construct a *stack-allocated* list instead of a heap-allocated list in implementations that support such.

In the following example,

```
(let ((x (list 'a1 'b1 'c1))
      (y (cons 'a2 (cons 'b2 (cons 'c2 nil)))))
  (declare (dynamic-extent x y))
  ...)
```

The *otherwise inaccessible parts* of x are three *conses*, and the *otherwise inaccessible parts* of y are three other *conses*. None of the symbols a1, b1, c1, a2, b2, c2, or **nil** is an *otherwise inaccessible part* of x or y because each is *interned* and hence *accessible* by the *package* (or *packages*) in which it is *interned*. However, if a freshly allocated *uninterned symbol* had been used, it would have been an *otherwise inaccessible part* of the *list* which contained it.

```
;; In this example, the implementation is permitted to stack allocate
;; the list that is bound to X.
(let ((x (list 1 2 3)))
  (declare (dynamic-extent x))
  (print x)
  :done)
▷ (1 2 3)
→ :DONE

;; In this example, the list to be bound to L can be stack-allocated.
(defun zap (x y z)
  (do ((l (list x y z) (cdr l)))
      ((null l))
    (declare (dynamic-extent l))
    (prin1 (car l)))) → ZAP
(zap 1 2 3)
▷ 123
→ NIL

;; Some implementations might open-code LIST-ALL-PACKAGES in a way
;; that permits using stack allocation of the list to be bound to L.
(do ((l (list-all-packages) (cdr l)))
    ((null l))
  (declare (dynamic-extent l))
  (let ((name (package-name (car l))))
    (when (string-search "COMMON-LISP" name) (print name))))
```

```
▷ "COMMON-LISP"
▷ "COMMON-LISP-USER"
→ NIL

;; Some implementations might have the ability to stack allocate
;; rest lists.  A declaration such as the following should be a cue
;; to such implementations that stack-allocation of the rest list
;; would be desirable.
 (defun add (&rest x)
   (declare (dynamic-extent x))
   (apply #'+ x)) → ADD
 (add 1 2 3) → 6

 (defun zap (n m)
   ;; Computes (RANDOM (+ M 1)) at relative speed of roughly O(N).
   ;; It may be slow, but with a good compiler at least it
   ;; doesn't waste much heap storage.  :-}
   (let ((a (make-array n)))
     (declare (dynamic-extent a))
     (dotimes (i n)
       (declare (dynamic-extent i))
       (setf (aref a i) (random (+ i 1))))
     (aref a m))) → ZAP
 (< (zap 5 3) 3) → true
```

The following are in error, since the value of `x` is used outside of its *extent*:

```
 (length (list (let ((x (list 1 2 3)))  ; Invalid
                (declare (dynamic-extent x))
                x)))

 (progn (let ((x (list 1 2 3)))  ; Invalid
          (declare (dynamic-extent x))
          x)
        nil)
```

See Also:

declare

Notes:

The most common optimization is to *stack allocate* the initial value of the *objects* named by the *vars*.

It is permissible for an implementation to simply ignore this declaration.

type *Declaration*

Syntax:

(type *typespec* {*var*}*)

(*typespec* {*var*}*)

Arguments:

typespec—a *type specifier*.

var—a *variable name*.

Valid Context:

declaration or *proclamation*

Binding Types Affected:

variable

Description:

Affects only variable *bindings* and specifies that the *vars* take on values only of the specified *typespec*. In particular, values assigned to the variables by **setq**, as well as the initial values of the *vars* must be of the specified *typespec*. **type** declarations never apply to function *bindings* (see **ftype**).

A type declaration of a *symbol* defined by **symbol-macrolet** is equivalent to wrapping a **the** expression around the expansion of that *symbol*, although the *symbol*'s *macro expansion* is not actually affected.

The meaning of a type declaration is equivalent to changing each reference to a variable (*var*) within the scope of the declaration to (the *typespec var*), changing each expression assigned to the variable (*new-value*) within the scope of the declaration to (the *typespec new-value*), and executing (the *typespec var*) at the moment the scope of the declaration is entered.

A *type* declaration is valid in all declarations. The interpretation of a type declaration is as follows:

1. During the execution of any reference to the declared variable within the scope of the declaration, the consequences are undefined if the value of the declared variable is not of the declared *type*.

2. During the execution of any **setq** of the declared variable within the scope of the declaration, the consequences are undefined if the newly assigned value of the declared variable is not of the declared *type*.

3. At the moment the scope of the declaration is entered, the consequences are undefined if

the value of the declared variable is not of the declared *type*.

A *type* declaration affects only variable references within its scope.

If nested *type* declarations refer to the same variable, then the value of the variable must be a member of the intersection of the declared *types*.

If there is a local `type` declaration for a dynamic variable, and there is also a global `type` proclamation for that same variable, then the value of the variable within the scope of the local declaration must be a member of the intersection of the two declared *types*.

type declarations can be *free declarations* or *bound declarations*.

A *symbol* cannot be both the name of a *type* and the name of a declaration. Defining a *symbol* as the *name* of a *class*, *structure*, *condition*, or *type*, when the *symbol* has been *declared* as a declaration name, or vice versa, signals an error.

Within the *lexical scope* of an **array** type declaration, all references to *array elements* are assumed to satisfy the *expressed array element type* (as opposed to the *upgraded array element type*). A compiler can treat the code within the scope of the **array** type declaration as if each *access* of an *array element* were surrounded by an appropriate **the** form.

Examples:

```
(defun f (x y)
  (declare (type fixnum x y))
  (let ((z (+ x y)))
    (declare (type fixnum z))
    z)) → F
(f 1 2) → 3
;; The previous definition of F is equivalent to
(defun f (x y)
  ;; This declaration is a shorthand form of the TYPE declaration
  (declare (fixnum x y))
  ;; To declare the type of a return value, it's not necessary to
  ;; create a named variable.  A THE special form can be used instead.
  (the fixnum (+ x y))) → F
(f 1 2) → 3


(defvar *one-array* (make-array 10 :element-type '(signed-byte 5)))
(defvar *another-array* (make-array 10 :element-type '(signed-byte 8)))

(defun frob (an-array)
  (declare (type (array (signed-byte 5) 1) an-array))
  (setf (aref an-array 1) 31)
  (setf (aref an-array 2) 127)
```



```
  (setf (aref an-array 3) (* 2 (aref an-array 3)))
  (let ((foo 0))
    (declare (type (signed-byte 5) foo))
    (setf foo (aref an-array 0))))

(frob *one-array*)
(frob *another-array*)
```

The above definition of `frob` is equivalent to:

```
(defun frob (an-array)
  (setf (the (signed-byte 5) (aref an-array 1)) 31)
  (setf (the (signed-byte 5) (aref an-array 2)) 127)
  (setf (the (signed-byte 5) (aref an-array 3))
        (* 2 (the (signed-byte 5) (aref an-array 3))))
  (let ((foo 0))
    (declare (type (signed-byte 5) foo))
    (setf foo (the (signed-byte 5) (aref an-array 0)))))
```

Given an implementation in which *fixnums* are 29 bits but **fixnum** *arrays* are upgraded to signed 32-bit *arrays*, the following could be compiled with all *fixnum* arithmetic:

```
(defun bump-counters (counters)
  (declare (type (array fixnum *) bump-counters))
  (dotimes (i (length counters))
    (incf (aref counters i))))
```

See Also:

declare, **declaim**, **proclaim**

Notes:

(*typespec* {*var*}*) is an abbreviation for (`type` *typespec* {*var*}*).

A **type** declaration for the arguments to a function does not necessarily imply anything about the type of the result. The following function is not permitted to be compiled using *implementation-dependent fixnum*-only arithmetic:

```
(defun f (x y) (declare (fixnum x y)) (+ x y))
```

To see why, consider `(f most-positive-fixnum 1)`. Common Lisp defines that `F` must return a *bignum* here, rather than signal an error or produce a mathematically incorrect result. If you have special knowledge such "*fixnum* overflow" cases will not come up, you can declare the result value to be in the *fixnum* range, enabling some compilers to use more efficient arithmetic:

```
(defun f (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Note, however, that in the three-argument case, because of the possibility of an implicit intermediate value growing too large, the following will not cause *implementation-dependent fixnum*-only arithmetic to be used:

```
(defun f (x y)
  (declare (fixnum x y z))
  (the fixnum (+ x y z)))
```

To see why, consider `(f most-positive-fixnum 1 -1)`. Although the arguments and the result are all *fixnums*, an intermediate value is not a *fixnum*. If it is important that *implementation-dependent fixnum*-only arithmetic be selected in *implementations* that provide it, consider writing something like this instead:

```
(defun f (x y)
  (declare (fixnum x y z))
  (the fixnum (+ (the fixnum (+ x y)) z)))
```

inline, notinline *Declaration*

Syntax:

(inline {*function-name*}*)

(notinline {*function-name*}*)

Arguments:

function-name—a *function name*.

Valid Context:

declaration or *proclamation*

Binding Types Affected:

function

Description:

inline specifies that it is desirable for the compiler to produce inline calls to the *functions* named by *function-names*; that is, the code for a specified *function-name* should be integrated into the calling routine, appearing "in line" in place of a procedure call. A compiler is free to ignore this declaration. **inline** declarations never apply to variable *bindings*.

If one of the *functions* mentioned has a lexically apparent local definition (as made by **flet** or **labels**), then the declaration applies to that local definition and not to the global function definition.

While no *conforming implementation* is required to perform inline expansion of user-defined functions, those *implementations* that do attempt to recognize the following paradigm:

To define a *function* `f` that is not **inline** by default but for which `(declare (inline f))` will make *f* be locally inlined, the proper definition sequence is:

```
(declaim (inline f))
(defun f ...)
(declaim (notinline f))
```

The **inline** proclamation preceding the **defun** *form* ensures that the *compiler* has the opportunity save the information necessary for inline expansion, and the **notinline** proclamation following the **defun** *form* prevents `f` from being expanded inline everywhere.

notinline specifies that it is undesirable to compile the *functions* named by *function-names* in-line. A compiler is not free to ignore this declaration; calls to the specified functions must be implemented as out-of-line subroutine calls.

If one of the *functions* mentioned has a lexically apparent local definition (as made by **flet** or **labels**), then the declaration applies to that local definition and not to the global function definition.

In the presence of a *compiler macro* definition for *function-name*, a **notinline** declaration prevents that *compiler macro* from being used. An **inline** declaration may be used to encourage use of *compiler macro* definitions. **inline** and **notinline** declarations otherwise have no effect when the lexically visible definition of *function-name* is a *macro* definition.

inline and **notinline** declarations can be *free declarations* or *bound declarations*. **inline** and **notinline** declarations of functions that appear before the body of a **flet** or **labels** *form* that defines that function are *bound declarations*. Such declarations in other contexts are *free declarations*.

Examples:

```
;; The globally defined function DISPATCH should be open-coded,
;; if the implementation supports inlining, unless a NOTINLINE
;; declaration overrides this effect.
(declaim (inline dispatch))
(defun dispatch (x) (funcall (get (car x) 'dispatch) x))
;; Here is an example where inlining would be encouraged.
(defun top-level-1 () (dispatch (read-command)))
;; Here is an example where inlining would be prohibited.
(defun top-level-2 ()
  (declare (notinline dispatch))
  (dispatch (read-command)))
;; Here is an example where inlining would be prohibited.
(declaim (notinline dispatch))
(defun top-level-3 () (dispatch (read-command)))
```

```
;; Here is an example where inlining would be encouraged.
(defun top-level-4 ()
  (declare (inline dispatch))
  (dispatch (read-command)))
```

See Also:

declare, **declaim**, **proclaim**

ftype *Declaration*

Syntax:

(ftype *type* {*function-name*}*)

Arguments:

function-name—a *function name*.

type—a *type specifier*.

Valid Context:

declaration or *proclamation*

Binding Types Affected:

function

Description:

Specifies that the *functions* named by *function-names* are of the functional type *type*. For example:

```
(declare (ftype (function (integer list) t) ith)
         (ftype (function (number) float) sine cosine))
```

If one of the *functions* mentioned has a lexically apparent local definition (as made by **flet** or **labels**), then the declaration applies to that local definition and not to the global function definition. **ftype** declarations never apply to variable *bindings* (see type).

The lexically apparent bindings of *function-names* must not be *macro* definitions. (This is because **ftype** declares the functional definition of each *function name* to be of a particular subtype of **function**, and *macros* do not denote *functions*.)

ftype declarations can be *free declarations* or *bound declarations*. **ftype** declarations of functions that appear before the body of a **flet** or **labels** *form* that defines that function are *bound declarations*. Such declarations in other contexts are *free declarations*.

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

See Also:

declare, **declaim**, **proclaim**

declaration *Declaration*

Syntax:

(declaration {*name*}*)

Arguments:

name—a *symbol*.

Valid Context:

proclamation only

Description:

Advises the compiler that each *name* is a valid but potentially non-standard declaration name. The purpose of this is to tell one compiler not to issue warnings for declarations meant for another compiler or other program processor.

Examples:

```
(declaim (declaration author target-language target-machine))
(declaim (target-language ada))
(declaim (target-machine IBM-650))
(defun strangep (x)
  (declare (author "Harry Tweeker"))
  (member x '(strange weird odd peculiar)))
```

See Also:

declaim, **proclaim**

optimize *Declaration*

Syntax:

(optimize {*quality* | (*quality value*)}*)

Arguments:

quality—an *optimize quality*.

value—one of the *integers* 0, 1, 2, or 3.

Valid Context:

declaration or *proclamation*

Description:

Advises the compiler that each *quality* should be given attention according to the specified corresponding *value*. Each *quality* must be a *symbol* naming an *optimize quality*; the names and meanings of the standard *optimize qualities* are shown in Figure 3–25.

Name	Meaning
compilation-speed	speed of the compilation process
debug	ease of debugging
safety	run-time error checking
space	both code size and run-time space
speed	speed of the object code

Figure 3–25. Optimize qualities

There may be other, *implementation-defined optimize qualities*.

A *value* 0 means that the corresponding *quality* is totally unimportant, and 3 that the *quality* is extremely important; 1 and 2 are intermediate values, with 1 the neutral value. (*quality* 3) can be abbreviated to *quality*.

Note that *code* which has the optimization (safety 3), or just **safety**, is called *safe code*.

The consequences are unspecified if a *quality* appears more than once with *different values*.

Examples:

```
(defun often-used-subroutine (x y)
  (declare (optimize (safety 2)))
  (error-check x y)
  (hairy-setup x)
  (do ((i 0 (+ i 1))
       (z x (cdr z)))
      ((null z))
    ;; This inner loop really needs to burn.
    (declare (optimize speed))
    (declare (fixnum i))
    ))
```

See Also:

declare, **declaim**, **proclaim**, Section 3.3.4 (Declaration Scope)

Notes:

An **optimize** declaration never applies to either a *variable* or a *function binding*. An **optimize** declaration can only be a *free declaration*. For more information, see Section 3.3.4 (Declaration Scope).

special *Declaration*

Syntax:

(special {*var*}*)

Arguments:

var—a *symbol*.

Valid Context:

declaration or *proclamation*

Binding Types Affected:

variable

Description:

Specifies that all of the *vars* named are dynamic. This specifier affects variable *bindings* and affects references. All variable *bindings* affected are made to be dynamic *bindings*, and affected variable references refer to the current dynamic *binding*. For example:

```
(defun hack (thing *mod*)    ;The binding of the parameter
  (declare (special *mod*))  ; *mod* is visible to hack1,
  (hack1 (car thing)))       ; but not that of thing.
(defun hack1 (arg)
  (declare (special *mod*))  ;Declare references to *mod*
                             ;within hack1 to be special.
  (if (atom arg) *mod*
      (cons (hack1 (car arg)) (hack1 (cdr arg)))))
```

special

A **special** declaration does not affect inner *bindings* of a *var*; the inner *bindings* implicitly shadow a **special** declaration and must be explicitly re-declared to be **special**. **special** declarations never apply to function *bindings*.

special declarations can be either *bound declarations*, affecting both a binding and references, or *free declarations*, affecting only references, depending on whether the declaration is attached to a variable binding.

When used in a *proclamation*, a **special** *declaration specifier* applies to all *bindings* as well as to all references of the mentioned variables. For example, after

```
(declaim (special x))
```

then in a function definition such as

```
(defun example (x) ...)
```

the parameter x is bound as a dynamic variable rather than as a lexical variable.

Examples:

```
(defun declare-eg (y)                 ;this y is special
 (declare (special y))
 (let ((y t))                         ;this y is lexical
      (list y
            (locally (declare (special y)) y)))) ;this y refers to the
                                                 ;special binding of y
→ DECLARE-EG
 (declare-eg nil) → (T NIL)

(setf (symbol-value 'x) 6)
(defun foo (x)                         ;a lexical binding of x
  (print x)
  (let ((x (1+ x)))                    ;a special binding of x
    (declare (special x))              ;and a lexical reference
    (bar))
  (1+ x))
(defun bar ()
  (print (locally (declare (special x))
           x)))
(foo 10)
▷ 10
▷ 11
→ 11

(setf (symbol-value 'x) 6)
(defun bar (x y)            ;[1] 1st occurrence of x
  (let ((old-x x)           ;[2] 2nd occurrence of x -- same as 1st occurrence
        (x y))              ;[3] 3rd occurrence of x
    (declare (special x))
    (list old-x x)))
(bar 'first 'second) → (FIRST SECOND)
```

```
 (defun few (x &optional (y *foo*))
   (declare (special *foo*))
   ...)
```

The reference to *foo* in the first line of this example is not **special** even though there is a **special** declaration in the second line.

```
 (declaim (special prosp)) → implementation-dependent
 (setq prosp 1 reg 1) → 1
 (let ((prosp 2) (reg 2))         ;the binding of prosp is special
    (set 'prosp 3) (set 'reg 3)   ;due to the preceding proclamation,
    (list prosp reg))             ;whereas the variable reg is lexical
→ (3 2)
 (list prosp reg) → (1 3)

 (declaim (special x))          ;x is always special.
 (defun example (x y)
   (declare (special y))
   (let ((y 3) (x (* x 2)))
     (print (+ y (locally (declare (special y)) y)))
     (let ((y 4)) (declare (special y)) (foo x)))) → EXAMPLE
```

In the contorted code above, the outermost and innermost *bindings* of y are dynamic, but the middle binding is lexical. The two arguments to + are different, one being the value, which is 3, of the lexical variable y, and the other being the value of the dynamic variable named y (a *binding* of which happens, coincidentally, to lexically surround it at an outer level). All the *bindings* of x and references to x are dynamic, however, because of the proclamation that x is always **special**.

See Also:

defparameter, **defvar**

locally *Special Operator*

Syntax:

locally {*declaration*}* {*form*}* → {*result*}*

Arguments and Values:

Declaration—a **declare** *expression*; not evaluated.

forms—an *implicit progn*.

results—the *values* of the *forms*.

Description:

Sequentially evaluates a body of *forms* in a *lexical environment* where the given *declarations* have effect.

Examples:

```
 (defun sample-function (y)  ;this y is regarded as special
   (declare (special y))
   (let ((y t))              ;this y is regarded as lexical
     (list y
           (locally (declare (special y))
             ;; this next y is regarded as special
             y))))
→ SAMPLE-FUNCTION
 (sample-function nil) → (T NIL)
 (setq x '(1 2 3) y '(4 . 5)) → (4 . 5)

;;; The following declarations are not notably useful in specific.
;;; They just offer a sample of valid declaration syntax using LOCALLY.
 (locally (declare (inline floor) (notinline car cdr))
          (declare (optimize space))
    (floor (car x) (cdr y))) → 0, 1


;;; This example shows a definition of a function that has a particular set
;;; of OPTIMIZE settings made locally to that definition.
 (locally (declare (optimize (safety 3) (space 3) (speed 0)))
   (defun frob (w x y &optional (z (foo x y)))
     (mumble x y z w)))
→ FROB

;;; This is like the previous example, except that the optimize settings
;;; remain in effect for subsequent definitions in the same compilation unit.
 (declaim (optimize (safety 3) (space 3) (speed 0)))
 (defun frob (w x y &optional (z (foo x y)))
   (mumble x y z w))
→ FROB
```

See Also:

declare

Notes:

The **special** declaration may be used with **locally** to affect references to, rather than *bindings* of, *variables*.

If a **locally** *form* is a *top level form*, the body *forms* are also processed as *top level forms*. See Section 3.2.3 (File Compilation).

the *Special Operator*

Syntax:

the *value-type form* → {*result*}*

Arguments and Values:

value-type—a *type specifier*; not evaluated.

form—a *form*; evaluated.

results—the *values* resulting from the *evaluation* of *form*. These *values* must conform to the *type* supplied by *value-type*; see below.

Description:

the specifies that the $values_{1a}$ returned by *form* are of the *types* specified by *value-type*. The consequences are undefined if any *result* is not of the declared type.

It is permissible for *form* to *yield* a different number of *values* than are specified by *value-type*, provided that the values for which *types* are declared are indeed of those *types*. Missing values are treated as **nil** for the purposes of checking their *types*.

Regardless of number of *values* declared by *value-type*, the number of *values* returned by the **the** *special form* is the same as the number of *values* returned by *form*.

Examples:

```
 (the symbol (car (list (gensym)))) → #:G9876
 (the fixnum (+ 5 7)) → 12
 (the (values) (truncate 3.2 2)) → 1, 1.2
 (the integer (truncate 3.2 2)) → 1, 1.2
 (the (values integer) (truncate 3.2 2)) → 1, 1.2
 (the (values integer float) (truncate 3.2 2))   → 1, 1.2
 (the (values integer float symbol) (truncate 3.2 2)) → 1, 1.2
 (the (values integer float symbol t null list)
      (truncate 3.2 2)) → 1, 1.2
 (let ((i 100))
    (declare (fixnum i))
```

```
   (the fixnum (1+ i))) → 101
(let* ((x (list 'a 'b 'c))
       (y 5))
   (setf (the fixnum (car x)) y)
   x) → (5 B C)
```

Exceptional Situations:

The consequences are undefined if the *values yielded* by the *form* are not of the *type* specified by *value-type*.

See Also:

values

Notes:

The **values** *type specifier* can be used to indicate the types of *multiple values*:

```
(the (values integer integer) (floor x y))
(the (values string t)
     (gethash the-key the-string-table))
```

setf can be used with **the** type declarations. In this case the declaration is transferred to the form that specifies the new value. The resulting **setf** *form* is then analyzed.

special-operator-p *Function*

Syntax:

special-operator-p *symbol* → *generalized-boolean*

Arguments and Values:

symbol—a *symbol*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *symbol* is a *special operator*; otherwise, returns *false*.

Examples:

```
(special-operator-p 'if) → true
(special-operator-p 'car) → false
(special-operator-p 'one) → false
```


Exceptional Situations:

Should signal **type-error** if its argument is not a *symbol*.

Notes:

Historically, this function was called `special-form-p`. The name was finally declared a misnomer and changed, since it returned true for *special operators*, not *special forms*.

constantp *Function*

Syntax:

constantp *form* `&optional` *environment* → *generalized-boolean*

Arguments and Values:

form—a *form*.

environment—an *environment object*. The default is **nil**.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *form* can be determined by the *implementation* to be a *constant form* in the indicated *environment*; otherwise, it returns *false* indicating either that the *form* is not a *constant form* or that it cannot be determined whether or not *form* is a *constant form*.

The following kinds of *forms* are considered *constant forms*:

- *Self-evaluating objects* (such as *numbers*, *characters*, and the various kinds of *arrays*) are always considered *constant forms* and must be recognized as such by **constantp**.

- *Constant variables*, such as *keywords*, symbols defined by Common Lisp as constant (such as **nil**, **t**, and **pi**), and symbols declared as constant by the user in the indicated *environment* using **defconstant** are always considered *constant forms* and must be recognized as such by **constantp**.

- **quote** *forms* are always considered *constant forms* and must be recognized as such by **constantp**.

- An *implementation* is permitted, but not required, to detect additional *constant forms*. If it does, it is also permitted, but not required, to make use of information in the *environment*. Examples of *constant forms* for which **constantp** might or might not return *true* are: `(sqrt pi)`, `(+ 3 2)`, `(length '(a b c))`, and `(let ((x 7)) (zerop x))`.

constantp

If an *implementation* chooses to make use of the *environment* information, such actions as expanding *macros* or performing function inlining are permitted to be used, but not required; however, expanding *compiler macros* is not permitted.

Examples:

```
(constantp 1) → true
(constantp 'temp) → false
(constantp ''temp)) → true
(defconstant this-is-a-constant 'never-changing) → THIS-IS-A-CONSTANT
(constantp 'this-is-a-constant) → true
(constantp "temp") → true
(setq a 6) → 6
(constantp a) → true
(constantp '(sin pi)) → implementation-dependent
(constantp '(car '(x))) → implementation-dependent
(constantp '(eql x x)) → implementation-dependent
(constantp '(typep x 'nil)) → implementation-dependent
(constantp '(typep x 't)) → implementation-dependent
(constantp '(values this-is-a-constant)) → implementation-dependent
(constantp '(values 'x 'y)) → implementation-dependent
(constantp '(let ((a '(a b c))) (+ (length a) 6))) → implementation-dependent
```

Affected By:

The state of the global environment (*e.g.*, which *symbols* have been declared to be the *names* of *constant variables*).

See Also:

defconstant

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