```
;;; 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.

declare

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*.

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*.

dynamic-extent

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*.

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.

inline, notinline

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*.

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`.

optimize

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)))))
```

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))
```

special

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))`.

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

```
(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