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

Programming Language—Common Lisp

5. Data and Control Flow

5.1 Generalized Reference

5.1.1 Overview of Places and Generalized Reference

A ***generalized reference*** is the use of a *form*, sometimes called a ***place***, as if it were a *variable* that could be read and written. The *value* of a *place* is the *object* to which the *place form* evaluates. The *value* of a *place* can be changed by using **setf**. The concept of binding a *place* is not defined in Common Lisp, but an *implementation* is permitted to extend the language by defining this concept.

Figure 5–1 contains examples of the use of **setf**. Note that the values returned by evaluating the *forms* in column two are not necessarily the same as those obtained by evaluating the *forms* in column three. In general, the exact *macro expansion* of a **setf** *form* is not guaranteed and can even be *implementation-dependent*; all that is guaranteed is that the expansion is an update form that works for that particular *implementation*, that the left-to-right evaluation of *subforms* is preserved, and that the ultimate result of evaluating **setf** is the value or values being stored.

Access function	Update Function	Update using setf
`x`	`(setq x datum)`	`(setf x datum)`
`(car x)`	`(rplaca x datum)`	`(setf (car x) datum)`
`(symbol-value x)`	`(set x datum)`	`(setf (symbol-value x) datum)`

Figure 5–1. Examples of setf

Figure 5–2 shows *operators* relating to *places* and *generalized reference*.

assert	**defsetf**	**push**
ccase	**get-setf-expansion**	**remf**
ctypecase	**getf**	**rotatef**
decf	**incf**	**setf**
define-modify-macro	**pop**	**shiftf**
define-setf-expander	**psetf**	

Figure 5–2. Operators relating to places and generalized reference.

Some of the *operators* above manipulate *places* and some manipulate *setf expanders*. A *setf expansion* can be derived from any *place*. New *setf expanders* can be defined by using **defsetf** and **define-setf-expander**.

5.1.1.1 Evaluation of Subforms to Places

The following rules apply to the *evaluation* of *subforms* in a *place*:

1. The evaluation ordering of *subforms* within a *place* is determined by the order specified by the second value returned by **get-setf-expansion**. For all *places* defined by this specification (*e.g.*, **getf**, **ldb**, ...), this order of evaluation is left-to-right. When a *place* is derived from a macro expansion, this rule is applied after the macro is expanded to find the appropriate *place*.

   Places defined by using **defmacro** or **define-setf-expander** use the evaluation order defined by those definitions. For example, consider the following:

   ```
   (defmacro wrong-order (x y) `(getf ,y ,x))
   ```

   This following *form* evaluates `place2` first and then `place1` because that is the order they are evaluated in the macro expansion:

   ```
   (push value (wrong-order place1 place2))
   ```

2. For the *macros* that manipulate *places* (**push**, **pushnew**, **remf**, **incf**, **decf**, **shiftf**, **rotatef**, **psetf**, **setf**, **pop**, and those defined by **define-modify-macro**) the *subforms* of the macro call are evaluated exactly once in left-to-right order, with the *subforms* of the *places* evaluated in the order specified in (1).

   push, **pushnew**, **remf**, **incf**, **decf**, **shiftf**, **rotatef**, **psetf**, **pop** evaluate all *subforms* before modifying any of the *place* locations. **setf** (in the case when **setf** has more than two arguments) performs its operation on each pair in sequence. For example, in

   ```
   (setf place1 value1 place2 value2 ...)
   ```

   the *subforms* of `place1` and `value1` are evaluated, the location specified by `place1` is modified to contain the value returned by `value1`, and then the rest of the **setf** form is processed in a like manner.

3. For **check-type**, **ctypecase**, and **ccase**, *subforms* of the *place* are evaluated once as in (1), but might be evaluated again if the type check fails in the case of **check-type** or none of the cases hold in **ctypecase** and **ccase**.

4. For **assert**, the order of evaluation of the generalized references is not specified.

Rules 2, 3 and 4 cover all *standardized macros* that manipulate *places*.

5.1.1.1.1 Examples of Evaluation of Subforms to Places

```
(let ((ref2 (list '())))
  (push (progn (princ "1") 'ref-1)
```

```
          (car (progn (princ "2") ref2))))
▷ 12
→ (REF1)

 (let (x)
    (push (setq x (list 'a))
          (car (setq x (list 'b))))
     x)
→ (((A) . B))
```

push first evaluates `(setq x (list 'a))` → `(a)`, then evaluates `(setq x (list 'b))` → `(b)`, then modifies the *car* of this latest value to be `((a) . b)`.

5.1.1.2 Setf Expansions

Sometimes it is possible to avoid evaluating *subforms* of a *place* multiple times or in the wrong order. A *setf expansion* for a given access form can be expressed as an ordered collection of five *objects*:

List of temporary variables

a list of symbols naming temporary variables to be bound sequentially, as if by **let***, to *values* resulting from value forms.

List of value forms

a list of forms (typically, *subforms* of the *place*) which when evaluated yield the values to which the corresponding temporary variables should be bound.

List of store variables

a list of symbols naming temporary store variables which are to hold the new values that will be assigned to the *place*.

Storing form

a form which can reference both the temporary and the store variables, and which changes the *value* of the *place* and guarantees to return as its values the values of the store variables, which are the correct values for **setf** to return.

Accessing form

a *form* which can reference the temporary variables, and which returns the *value* of the *place*.

The value returned by the accessing form is affected by execution of the storing form, but either of these forms might be evaluated any number of times.

It is possible to do more than one **setf** in parallel via **psetf**, **shiftf**, and **rotatef**. Because of this, the *setf expander* must produce new temporary and store variable names every time. For examples of how to do this, see **gensym**.

For each *standardized* accessor function *F*, unless it is explicitly documented otherwise, it is *implementation-dependent* whether the ability to use an *F form* as a **setf** *place* is implemented by a *setf expander* or a *setf function*. Also, it follows from this that it is *implementation-dependent* whether the name `(setf F)` is *fbound*.

5.1.1.2.1 Examples of Setf Expansions

Examples of the contents of the constituents of *setf expansions* follow.

For a variable *x*:

```
()                                   ;list of temporary variables
()                                   ;list of value forms
(g0001)                              ;list of store variables
(setq x g0001)                       ;storing form
x                                    ;accessing form
```

Figure 5–3. Sample Setf Expansion of a Variable

For `(car exp)`:

```
(g0002)                                     ;list of temporary variables
(exp)                                       ;list of value forms
(g0003)                                     ;list of store variables
(progn (rplaca g0002 g0003) g0003)          ;storing form
(car g0002)                                 ;accessing form
```

Figure 5–4. Sample Setf Expansion of a CAR Form

For `(subseq seq s e)`:

```
(g0004 g0005 g0006)                  ;list of temporary variables
(seq s e)                            ;list of value forms
(g0007)                              ;list of store variables
(progn (replace g0004 g0007 :start1 g0005 :end1 g0006) g0007)
                                     ;storing form
(subseq g0004 g0005 g0006)           ; accessing form
```

Figure 5–5. Sample Setf Expansion of a SUBSEQ Form

In some cases, if a *subform* of a *place* is itself a *place*, it is necessary to expand the *subform* in order to compute some of the values in the expansion of the outer *place*. For `(ldb bs (car exp))`:

```
(g0001 g0002)                          ;list of temporary variables
(bs exp)                               ;list of value forms
(g0003)                                ;list of store variables
(progn (rplaca g0002 (dpb g0003 g0001 (car g0002))) g0003)
                                       ;storing form
(ldb g0001 (car g0002))                ; accessing form
```

Figure 5–6. Sample Setf Expansion of a LDB Form

5.1.2 Kinds of Places

Several kinds of *places* are defined by Common Lisp; this section enumerates them. This set can be extended by *implementations* and by *programmer code*.

5.1.2.1 Variable Names as Places

The name of a *lexical variable* or *dynamic variable* can be used as a *place*.

5.1.2.2 Function Call Forms as Places

A *function form* can be used as a *place* if it falls into one of the following categories:

- A function call form whose first element is the name of any one of the functions in Figure 5–7.

aref	**cdadr**	**get**
bit	**cdar**	**gethash**
caaaar	**cddaar**	**logical-pathname-translations**
caaadr	**cddadr**	**macro-function**
caaar	**cddar**	**ninth**
caadar	**cdddar**	**nth**
caaddr	**cddddr**	**readtable-case**
caadr	**cdddr**	**rest**
caar	**cddr**	**row-major-aref**
cadaar	**cdr**	**sbit**
cadadr	**char**	**schar**
cadar	**class-name**	**second**
caddar	**compiler-macro-function**	**seventh**
cadddr	**documentation**	**sixth**
caddr	**eighth**	**slot-value**
cadr	**elt**	**subseq**
car	**fdefinition**	**svref**
cdaaar	**fifth**	**symbol-function**
cdaadr	**fill-pointer**	**symbol-plist**
cdaar	**find-class**	**symbol-value**
cdadar	**first**	**tenth**
cdaddr	**fourth**	**third**

Figure 5–7. Functions that setf can be used with—1

In the case of **subseq**, the replacement value must be a *sequence* whose elements might be contained by the sequence argument to **subseq**, but does not have to be a *sequence* of the same *type* as the *sequence* of which the subsequence is specified. If the length of the replacement value does not equal the length of the subsequence to be replaced, then the shorter length determines the number of elements to be stored, as for **replace**.

- A function call form whose first element is the name of a selector function constructed by **defstruct**. The function name must refer to the global function definition, rather than a locally defined *function*.

- A function call form whose first element is the name of any one of the functions in Figure 5–8, provided that the supplied argument to that function is in turn a *place* form; in this case the new *place* has stored back into it the result of applying the supplied "update" function.

Function name	**Argument that is a *place***	**Update function used**
ldb	second	**dpb**
mask-field	second	**deposit-field**
getf	first	*implementation-dependent*

Figure 5–8. Functions that setf can be used with—2

During the **setf** expansion of these *forms*, it is necessary to call **get-setf-expansion** in order to figure out how the inner, nested generalized variable must be treated.

The information from **get-setf-expansion** is used as follows.

ldb

In a form such as:

`(setf (ldb` *byte-spec place-form*`)` *value-form*`)`

the place referred to by the *place-form* must always be both *read* and *written*; note that the update is to the generalized variable specified by *place-form*, not to any object of *type* **integer**.

Thus this **setf** should generate code to do the following:

1. Evaluate *byte-spec* (and bind it into a temporary variable).
2. Bind the temporary variables for *place-form*.
3. Evaluate *value-form* (and bind its value or values into the store variable).
4. Do the *read* from *place-form*.
5. Do the *write* into *place-form* with the given bits of the *integer* fetched in step 4 replaced with the value from step 3.

If the evaluation of *value-form* in step 3 alters what is found in *place-form*, such as setting different bits of *integer*, then the change of the bits denoted by *byte-spec* is to that altered *integer*, because step 4 is done after the *value-form* evaluation. Nevertheless, the evaluations required for *binding* the temporary variables are done in steps 1 and 2, and thus the expected left-to-right evaluation order is seen. For example:

```
 (setq integer #x69) → #x69
 (rotatef (ldb (byte 4 4) integer)
          (ldb (byte 4 0) integer))
 integer → #x96
;;; This example is trying to swap two independent bit fields
;;; in an integer.  Note that the generalized variable of
;;; interest here is just the (possibly local) program variable
;;; integer.
```

mask-field

This case is the same as **ldb** in all essential aspects.

getf

In a form such as:

(setf (getf *place-form ind-form*) *value-form*)

the place referred to by *place-form* must always be both *read* and *written*; note that the update is to the generalized variable specified by *place-form*, not necessarily to the particular *list* that is the property list in question.

Thus this **setf** should generate code to do the following:

1. Bind the temporary variables for *place-form*.
2. Evaluate *ind-form* (and bind it into a temporary variable).
3. Evaluate *value-form* (and bind its value or values into the store variable).
4. Do the *read* from *place-form*.
5. Do the *write* into *place-form* with a possibly-new property list obtained by combining the values from steps 2, 3, and 4. (Note that the phrase "possibly-new property list" can mean that the former property list is somehow destructively re-used, or it can mean partial or full copying of it. Since either copying or destructive re-use can occur, the treatment of the resultant value for the possibly-new property list must proceed as if it were a different copy needing to be stored back into the generalized variable.)

If the evaluation of *value-form* in step 3 alters what is found in *place-form*, such as setting a different named property in the list, then the change of the property denoted by *ind-form* is to that altered list, because step 4 is done after the *value-form* evaluation. Nevertheless, the evaluations required for *binding* the temporary variables are done in steps 1 and 2, and thus the expected left-to-right evaluation order is seen.

For example:

```
(setq s (setq r (list (list 'a 1 'b 2 'c 3)))) → ((a 1 b 2 c 3))
(setf (getf (car r) 'b)
      (progn (setq r nil) 6)) → 6
r → NIL
```

```
 s → ((A 1 B 6 C 3))
;;; Note that the (setq r nil) does not affect the actions of
;;; the SETF because the value of R had already been saved in
;;; a temporary variable as part of the step 1. Only the CAR
;;; of this value will be retrieved, and subsequently modified
;;; after the value computation.
```

5.1.2.3 VALUES Forms as Places

A **values** *form* can be used as a *place*, provided that each of its *subforms* is also a *place* form.

A form such as

`(setf (values` *place-1* ... *place-n*) *values-form*)

does the following:

1. The *subforms* of each nested *place* are evaluated in left-to-right order.
2. The *values-form* is evaluated, and the first store variable from each *place* is bound to its return values as if by **multiple-value-bind**.
3. If the *setf expansion* for any *place* involves more than one store variable, then the additional store variables are bound to **nil**.
4. The storing forms for each *place* are evaluated in left-to-right order.

The storing form in the *setf expansion* of **values** returns as *multiple values*$_2$ the values of the store variables in step 2. That is, the number of values returned is the same as the number of *place* forms. This may be more or fewer values than are produced by the *values-form*.

5.1.2.4 THE Forms as Places

A **the** *form* can be used as a *place*, in which case the declaration is transferred to the *newvalue* form, and the resulting **setf** is analyzed. For example,

```
 (setf (the integer (cadr x)) (+ y 3))
```

is processed as if it were

```
 (setf (cadr x) (the integer (+ y 3)))
```

5.1.2.5 APPLY Forms as Places

The following situations involving **setf** of **apply** must be supported:

- `(setf (apply #'aref` *array* {*subscript*}* *more-subscripts*`)` *new-element*`)`
- `(setf (apply #'bit` *array* {*subscript*}* *more-subscripts*`)` *new-element*`)`
- `(setf (apply #'sbit` *array* {*subscript*}* *more-subscripts*`)` *new-element*`)`

In all three cases, the *element* of *array* designated by the concatenation of *subscripts* and *more-subscripts* (*i.e.*, the same *element* which would be *read* by the call to *apply* if it were not part of a **setf** *form*) is changed to have the *value* given by *new-element*. For these usages, the function name (**aref**, **bit**, or **sbit**) must refer to the global function definition, rather than a locally defined *function*.

No other *standardized function* is required to be supported, but an *implementation* may define such support. An *implementation* may also define support for *implementation-defined operators*.

If a user-defined *function* is used in this context, the following equivalence is true, except that care is taken to preserve proper left-to-right evaluation of argument *subforms*:

```
(setf (apply #'name {arg}*) val)
≡ (apply #'(setf name) val {arg}*)
```

5.1.2.6 Setf Expansions and Places

Any *compound form* for which the *operator* has a *setf expander* defined can be used as a *place*. The *operator* must refer to the global function definition, rather than a locally defined *function* or *macro*.

5.1.2.7 Macro Forms as Places

A *macro form* can be used as a *place*, in which case Common Lisp expands the *macro form* as if by **macroexpand-1** and then uses the *macro expansion* in place of the original *place*. Such *macro expansion* is attempted only after exhausting all other possibilities other than expanding into a call to a function named `(setf` *reader*`)`.

5.1.2.8 Symbol Macros as Places

A reference to a *symbol* that has been *established* as a *symbol macro* can be used as a *place*. In this case, **setf** expands the reference and then analyzes the resulting *form*.

5.1.2.9 Other Compound Forms as Places

For any other *compound form* for which the *operator* is a *symbol* *f*, the **setf** *form* expands into a call to the *function* named `(setf f)`. The first *argument* in the newly constructed *function form* is *newvalue* and the remaining *arguments* are the remaining *elements* of *place*. This expansion occurs regardless of whether *f* or `(setf f)` is defined as a *function* locally, globally, or not at all. For example,

```
(setf (f arg1 arg2 ...) new-value)
```

expands into a form with the same effect and value as

```
 (let ((#:temp-1 arg1)          ;force correct order of evaluation
       (#:temp-2 arg2)
       ...
       (#:temp-0 new-value))
   (funcall (function (setf f)) #:temp-0 #:temp-1 #:temp-2...))
```

A *function* named `(setf f)` must return its first argument as its only value in order to preserve the semantics of **setf**.

5.1.3 Treatment of Other Macros Based on SETF

For each of the "read-modify-write" *operators* in Figure 5–9, and for any additional *macros* defined by the *programmer* using **define-modify-macro**, an exception is made to the normal rule of left-to-right evaluation of arguments. Evaluation of *argument forms* occurs in left-to-right order, with the exception that for the *place argument*, the actual *read* of the "old value" from that *place* happens after all of the *argument form evaluations*, and just before a "new value" is computed and *written* back into the *place*.

Specifically, each of these *operators* can be viewed as involving a *form* with the following general syntax:

(*operator* {*preceding-form*}* *place* {*following-form*}*)

The evaluation of each such *form* proceeds like this:

1. *Evaluate* each of the *preceding-forms*, in left-to-right order.
2. *Evaluate* the *subforms* of the *place*, in the order specified by the second value of the *setf expansion* for that *place*.
3. *Evaluate* each of the *following-forms*, in left-to-right order.
4. *Read* the old value from *place*.
5. Compute the new value.
6. Store the new value into *place*.

decf	**pop**	**pushnew**
incf	**push**	**remf**

Figure 5–9. Read-Modify-Write Macros

5.2 Transfer of Control to an Exit Point

When a transfer of control is initiated by **go**, **return-from**, or **throw** the following events occur in order to accomplish the transfer of control. Note that for **go**, the *exit point* is the *form* within the **tagbody** that is being executed at the time the **go** is performed; for **return-from**, the *exit point* is the corresponding **block** *form*; and for **throw**, the *exit point* is the corresponding **catch** *form*.

1. Intervening *exit points* are "abandoned" (*i.e.*, their *extent* ends and it is no longer valid to attempt to transfer control through them).

2. The cleanup clauses of any intervening **unwind-protect** clauses are evaluated.

3. Intervening dynamic *bindings* of **special** variables, *catch tags*, *condition handlers*, and *restarts* are undone.

4. The *extent* of the *exit point* being invoked ends, and control is passed to the target.

The extent of an exit being "abandoned" because it is being passed over ends as soon as the transfer of control is initiated. That is, event 1 occurs at the beginning of the initiation of the transfer of control. The consequences are undefined if an attempt is made to transfer control to an *exit point* whose *dynamic extent* has ended.

Events 2 and 3 are actually performed interleaved, in the order corresponding to the reverse order in which they were established. The effect of this is that the cleanup clauses of an **unwind-protect** see the same dynamic *bindings* of variables and *catch tags* as were visible when the **unwind-protect** was entered.

Event 4 occurs at the end of the transfer of control.

apply *Function*

Syntax:

apply *function* &rest *args*$^+$ → {*result*}*

Arguments and Values:

function—a *function designator*.

args—a *spreadable argument list designator*.

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

Description:

Applies the *function* to the *args*.

When the *function* receives its arguments via **&rest**, it is permissible (but not required) for the *implementation* to *bind* the *rest parameter* to an *object* that shares structure with the last argument to **apply**. Because a *function* can neither detect whether it was called via **apply** nor whether (if so) the last argument to **apply** was a *constant*, *conforming programs* must neither rely on the *list* structure of a *rest list* to be freshly consed, nor modify that *list* structure.

setf can be used with **apply** in certain circumstances; see Section 5.1.2.5 (APPLY Forms as Places).

Examples:

```
(setq f '+) → +
(apply f '(1 2)) → 3
(setq f #'-) → #<FUNCTION ->
(apply f '(1 2)) → -1
(apply #'max 3 5 '(2 7 3)) → 7
(apply 'cons '((+ 2 3) 4)) → ((+ 2 3) . 4)
(apply #'+ '()) → 0

(defparameter *some-list* '(a b c))
(defun strange-test (&rest x) (eq x *some-list*))
(apply #'strange-test *some-list*) → implementation-dependent

(defun bad-boy (&rest x) (rplacd x 'y))
(bad-boy 'a 'b 'c) has undefined consequences.
(apply #'bad-boy *some-list*) has undefined consequences.


(defun foo (size &rest keys &key double &allow-other-keys)
  (let ((v (apply #'make-array size :allow-other-keys t keys)))
    (if double (concatenate (type-of v) v v) v)))
```

```
(foo 4 :initial-contents '(a b c d) :double t)
  → #(A B C D A B C D)
```

See Also:

funcall, **fdefinition**, **function**, Section 3.1 (Evaluation), Section 5.1.2.5 (APPLY Forms as Places)

defun *Macro*

Syntax:

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

Arguments and Values:

function-name—a *function name*.

lambda-list—an *ordinary lambda list*.

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

documentation—a *string*; not evaluated.

forms—an *implicit progn*.

block-name—the *function block name* of the *function-name*.

Description:

Defines a new *function* named *function-name* in the *global environment*. The body of the *function* defined by **defun** consists of *forms*; they are executed as an *implicit progn* when the *function* is called. **defun** can be used to define a new *function*, to install a corrected version of an incorrect definition, to redefine an already-defined *function*, or to redefine a *macro* as a *function*.

defun implicitly puts a **block** named *block-name* around the body *forms* (but not the *forms* in the *lambda-list*) of the *function* defined.

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

Evaluating **defun** causes *function-name* to be a global name for the *function* specified by the *lambda expression*

```
(lambda lambda-list
  ⟦{declaration}* | documentation⟧
  (block block-name {form}*))
```

processed in the *lexical environment* in which **defun** was executed.

defun

(None of the arguments are evaluated at macro expansion time.)

defun is not required to perform any compile-time side effects. In particular, **defun** does not make the *function* definition available at compile time. An *implementation* may choose to store information about the *function* for the purposes of compile-time error-checking (such as checking the number of arguments on calls), or to enable the *function* to be expanded inline.

Examples:

```
 (defun recur (x)
  (when (> x 0)
    (recur (1- x)))) → RECUR
 (defun ex (a b &optional c (d 66) &rest keys &key test (start 0))
    (list a b c d keys test start)) → EX
 (ex 1 2) → (1 2 NIL 66 NIL NIL 0)
 (ex 1 2 3 4 :test 'equal :start 50)
→ (1 2 3 4 (:TEST EQUAL :START 50) EQUAL 50)
 (ex :test 1 :start 2) → (:TEST 1 :START 2 NIL NIL 0)

 ;; This function assumes its callers have checked the types of the
 ;; arguments, and authorizes the compiler to build in that assumption.
 (defun discriminant (a b c)
   (declare (number a b c))
   "Compute the discriminant for a quadratic equation."
   (- (* b b) (* 4 a c))) → DISCRIMINANT
 (discriminant 1 2/3 -2) → 76/9

 ;; This function assumes its callers have not checked the types of the
 ;; arguments, and performs explicit type checks before making any assumptions.
 (defun careful-discriminant (a b c)
   "Compute the discriminant for a quadratic equation."
   (check-type a number)
   (check-type b number)
   (check-type c number)
   (locally (declare (number a b c))
     (- (* b b) (* 4 a c)))) → CAREFUL-DISCRIMINANT
 (careful-discriminant 1 2/3 -2) → 76/9
```

See Also:

flet, **labels**, **block**, **return-from**, **declare**, **documentation**, Section 3.1 (Evaluation), Section 3.4.1 (Ordinary Lambda Lists), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

return-from can be used to return prematurely from a *function* defined by **defun**.

Additional side effects might take place when additional information (typically debugging information) about the function definition is recorded.

fdefinition *Accessor*

Syntax:

fdefinition *function-name* → *definition*

(**setf** (**fdefinition** *function-name*) *new-definition*)

Arguments and Values:

function-name—a *function name*. In the non-**setf** case, the *name* must be *fbound* in the *global environment*.

definition—Current global function definition named by *function-name*.

new-definition—a *function*.

Description:

fdefinition *accesses* the current global function definition named by *function-name*. The definition may be a *function* or may be an *object* representing a *special form* or *macro*. The value returned by **fdefinition** when **fboundp** returns true but the *function-name* denotes a *macro* or *special form* is not well-defined, but **fdefinition** does not signal an error.

Exceptional Situations:

Should signal an error of *type* **type-error** if *function-name* is not a *function name*.

An error of *type* **undefined-function** is signaled in the non-**setf** case if *function-name* is not *fbound*.

See Also:

fboundp, **fmakunbound**, **macro-function**, **special-operator-p**, **symbol-function**

Notes:

fdefinition cannot *access* the value of a lexical function name produced by **flet** or **labels**; it can *access* only the global function value.

setf can be used with **fdefinition** to replace a global function definition when the *function-name*'s function definition does not represent a *special form*. **setf** of **fdefinition** requires a *function* as the new value. It is an error to set the **fdefinition** of a *function-name* to a *symbol*, a *list*, or the value returned by **fdefinition** on the name of a *macro* or *special form*.

fboundp

fboundp *Function*

Syntax:

fboundp *name* → *generalized-boolean*

Pronunciation:

[ˌef ˈbau̇ndpē]

Arguments and Values:

name—a *function name*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *name* is *fbound*; otherwise, returns *false*.

Examples:

```
(fboundp 'car) → true
(fboundp 'nth-value) → false
(fboundp 'with-open-file) → true
(fboundp 'unwind-protect) → true
(defun my-function (x) x) → MY-FUNCTION
(fboundp 'my-function) → true
(let ((saved-definition (symbol-function 'my-function)))
  (unwind-protect (progn (fmakunbound 'my-function)
                         (fboundp 'my-function))
    (setf (symbol-function 'my-function) saved-definition)))
→ false
(fboundp 'my-function) → true
(defmacro my-macro (x) `',x) → MY-MACRO
(fboundp 'my-macro) → true
(fmakunbound 'my-function) → MY-FUNCTION
(fboundp 'my-function) → false
(flet ((my-function (x) x))
  (fboundp 'my-function)) → false
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *name* is not a *function name*.

See Also:

symbol-function, **fmakunbound**, **fdefinition**

Notes:

It is permissible to call **symbol-function** on any *symbol* that is *fbound*.

fboundp is sometimes used to "guard" an access to the *function cell*, as in: `(if (fboundp x) (symbol-function x))`

Defining a *setf expander* *F* does not cause the *setf function* `(setf F)` to become defined.

fmakunbound *Function*

Syntax:

fmakunbound *name* → *name*

Pronunciation:

[ˌ**ef**ˈ**makən**ˌ**baůnd**] or [ˌ**ef**ˈ**mākən**ˌ**baůnd**]

Arguments and Values:

name—a *function name*.

Description:

Removes the *function* or *macro* definition, if any, of *name* in the *global environment*.

Examples:

```
(defun add-some (x) (+ x 19)) → ADD-SOME
 (fboundp 'add-some) → true
 (flet ((add-some (x) (+ x 37)))
    (fmakunbound 'add-some)
    (add-some 1)) → 38
 (fboundp 'add-some) → false
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *name* is not a *function name*.

The consequences are undefined if *name* is a *special operator*.

See Also:

fboundp, **makunbound**

flet, labels, macrolet *Special Operator*

Syntax:

flet ({(*function-name lambda-list* ⟦{*local-declaration*}* | *local-documentation*⟧ {*local-form*}*)}*)
{*declaration*}* {*form*}*

→ {*result*}*

labels ({(*function-name lambda-list* ⟦{*local-declaration*}* | *local-documentation*⟧ {*local-form*}*)}*)
{*declaration*}* {*form*}*

→ {*result*}*

macrolet ({(*name lambda-list* ⟦{*local-declaration*}* | *local-documentation*⟧ {*local-form*}*)}*)
{*declaration*}* {*form*}*

→ {*result*}*

Arguments and Values:

function-name—a *function name*.

name—a *symbol*.

lambda-list—a *lambda list*; for **flet** and **labels**, it is an *ordinary lambda list*; for **macrolet**, it is a *macro lambda list*.

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

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

local-documentation—a *string*; not evaluated.

local-forms, *forms*—an *implicit progn*.

results—the *values* of the *forms*.

Description:

flet, **labels**, and **macrolet** define local *functions* and *macros*, and execute *forms* using the local definitions. *Forms* are executed in order of occurrence.

The body forms (but not the *lambda list*) of each *function* created by **flet** and **labels** and each *macro* created by **macrolet** are enclosed in an *implicit block* whose name is the *function block name* of the *function-name* or *name*, as appropriate.

The scope of the *declarations* between the list of local function/macro definitions and the body *forms* in **flet** and **labels** does not include the bodies of the locally defined *functions*, except that for **labels**, any **inline**, **notinline**, or **ftype** declarations that refer to the locally defined functions do apply to the local function bodies. That is, their *scope* is the same as the function name that they affect. The scope of these *declarations* does not include the bodies of the macro expander functions defined by **macrolet**.

flet

flet defines locally named *functions* and executes a series of *forms* with these definition *bindings*. Any number of such local *functions* can be defined.

The *scope* of the name *binding* encompasses only the body. Within the body of **flet**, *function-names* matching those defined by **flet** refer to the locally defined *functions* rather than to the global function definitions of the same name. Also, within the scope of **flet**, global *setf expander* definitions of the *function-name* defined by **flet** do not apply. Note that this applies to `(defsetf` *f* `...)`, not `(defmethod (setf` *f*`) ...)`.

The names of *functions* defined by **flet** are in the *lexical environment*; they retain their local definitions only within the body of **flet**. The function definition bindings are visible only in the body of **flet**, not the definitions themselves. Within the function definitions, local function names that match those being defined refer to *functions* or *macros* defined outside the **flet**. **flet** can locally *shadow* a global function name, and the new definition can refer to the global definition.

Any *local-documentation* is attached to the corresponding local *function* (if one is actually created) as a *documentation string*.

labels

labels is equivalent to **flet** except that the scope of the defined function names for **labels** encompasses the function definitions themselves as well as the body.

macrolet

macrolet establishes local *macro* definitions, using the same format used by **defmacro**.

Within the body of **macrolet**, global *setf expander* definitions of the *names* defined by the **macrolet** do not apply; rather, **setf** expands the *macro form* and recursively process the resulting *form*.

The macro-expansion functions defined by **macrolet** are defined in the *lexical environment* in which the **macrolet** form appears. Declarations and **macrolet** and **symbol-macrolet** definitions affect the local macro definitions in a **macrolet**, but the consequences are undefined if the local macro definitions reference any local *variable* or *function bindings* that are visible in that *lexical environment*.

Any *local-documentation* is attached to the corresponding local *macro function* as a *documentation string*.

flet, labels, macrolet

Examples:

```
(defun foo (x flag)
  (macrolet ((fudge (z)
                ;The parameters x and flag are not accessible
                ; at this point; a reference to flag would be to
                ; the global variable of that name.
                `(if flag (* ,z ,z) ,z)))
   ;The parameters x and flag are accessible here.
    (+ x
       (fudge x)
       (fudge (+ x 1)))))
≡
(defun foo (x flag)
  (+ x
     (if flag (* x x) x)
     (if flag (* (+ x 1) (+ x 1)) (+ x 1))))
```

after macro expansion. The occurrences of `x` and `flag` legitimately refer to the parameters of the function `foo` because those parameters are visible at the site of the macro call which produced the expansion.

```
(flet ((flet1 (n) (+ n n)))
   (flet ((flet1 (n) (+ 2 (flet1 n))))
     (flet1 2))) → 6

(defun dummy-function () 'top-level) → DUMMY-FUNCTION
(funcall #'dummy-function) → TOP-LEVEL
(flet ((dummy-function () 'shadow))
     (funcall #'dummy-function)) → SHADOW
(eq (funcall #'dummy-function) (funcall 'dummy-function))
→ true
(flet ((dummy-function () 'shadow))
  (eq (funcall #'dummy-function)
      (funcall 'dummy-function)))
→ false

(defun recursive-times (k n)
  (labels ((temp (n)
             (if (zerop n) 0 (+ k (temp (1- n))))))
    (temp n))) → RECURSIVE-TIMES
(recursive-times 2 3) → 6

(defmacro mlets (x &environment env)
   (let ((form `(babbit ,x)))
     (macroexpand form env))) → MLETS
```

```
 (macrolet ((babbit (z) `(+ ,z ,z))) (mlets 5)) → 10

 (flet ((safesqrt (x) (sqrt (abs x))))
  ;; The safesqrt function is used in two places.
   (safesqrt (apply #'+ (map 'list #'safesqrt '(1 2 3 4 5 6)))))
→ 3.291173

 (defun integer-power (n k)
   (declare (integer n))
   (declare (type (integer 0 *) k))
   (labels ((expt0 (x k a)
              (declare (integer x a) (type (integer 0 *) k))
              (cond ((zerop k) a)
                    ((evenp k) (expt1 (* x x) (floor k 2) a))
                    (t (expt0 (* x x) (floor k 2) (* x a)))))
            (expt1 (x k a)
              (declare (integer x a) (type (integer 0 *) k))
              (cond ((evenp k) (expt1 (* x x) (floor k 2) a))
                    (t (expt0 (* x x) (floor k 2) (* x a))))))
    (expt0 n k 1))) → INTEGER-POWER


 (defun example (y l)
   (flet ((attach (x)
            (setq l (append l (list x)))))
     (declare (inline attach))
     (dolist (x y)
       (unless (null (cdr x))
         (attach x)))
     l))

 (example '((a apple apricot) (b banana) (c cherry) (d) (e))
          '((1) (2) (3) (4 2) (5) (6 3 2)))
→ ((1) (2) (3) (4 2) (5) (6 3 2) (A APPLE APRICOT) (B BANANA) (C CHERRY))
```

See Also:

declare, **defmacro**, **defun**, **documentation**, **let**, Section 3.1 (Evaluation), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

It is not possible to define recursive *functions* with **flet**. **labels** can be used to define mutually recursive *functions*.

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

funcall *Function*

Syntax:

funcall *function* &rest *args* → {*result*}*

Arguments and Values:

function—a *function designator*.

args—*arguments* to the *function*.

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

Description:

funcall applies *function* to *args*. If *function* is a *symbol*, it is coerced to a *function* as if by finding its *functional value* in the *global environment*.

Examples:

```
 (funcall #'+ 1 2 3) → 6
 (funcall 'car '(1 2 3)) → 1
 (funcall 'position 1 '(1 2 3 2 1) :start 1) → 4
 (cons 1 2) → (1 . 2)
 (flet ((cons (x y) `(kons ,x ,y)))
   (let ((cons (symbol-function '+)))
     (funcall #'cons
              (funcall 'cons 1 2)
              (funcall cons 1 2))))
→ (KONS (1 . 2) 3)
```

Exceptional Situations:

An error of *type* **undefined-function** should be signaled if *function* is a *symbol* that does not have a global definition as a *function* or that has a global definition as a *macro* or a *special operator*.

See Also:

apply, **function**, Section 3.1 (Evaluation)

Notes:

```
 (funcall function arg1 arg2 ...)
 ≡ (apply function arg1 arg2 ... nil)
 ≡ (apply function (list arg1 arg2 ...))
```

The difference between **funcall** and an ordinary function call is that in the former case the *function* is obtained by ordinary *evaluation* of a *form*, and in the latter case it is obtained by the special interpretation of the function position that normally occurs.

function *Special Operator*

Syntax:

function *name* → *function*

Arguments and Values:

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

function—a *function object*.

Description:

The *value* of **function** is the *functional value* of *name* in the current *lexical environment*.

If *name* is a *function name*, the functional definition of that name is that established by the innermost lexically enclosing **flet**, **labels**, or **macrolet** *form*, if there is one. Otherwise the global functional definition of the *function name* is returned.

If *name* is a *lambda expression*, then a *lexical closure* is returned. In situations where a *closure* over the same set of *bindings* might be produced more than once, the various resulting *closures* might or might not be **eq**.

It is an error to use **function** on a *function name* that does not denote a *function* in the lexical environment in which the **function** form appears. Specifically, it is an error to use **function** on a *symbol* that denotes a *macro* or *special form*. An implementation may choose not to signal this error for performance reasons, but implementations are forbidden from defining the failure to signal an error as a useful behavior.

Examples:

```
(defun adder (x) (function (lambda (y) (+ x y))))
```

The result of `(adder 3)` is a function that adds `3` to its argument:

```
(setq add3 (adder 3))
(funcall add3 5) → 8
```

This works because **function** creates a *closure* of the *lambda expression* that is able to refer to the *value* `3` of the variable `x` even after control has returned from the function `adder`.

See Also:

defun, **fdefinition**, **flet**, **labels**, **symbol-function**, Section 3.1.2.1.1 (Symbols as Forms), Section 2.4.8.2 (Sharpsign Single-Quote), Section 22.1.3.13 (Printing Other Objects)

Notes:

The notation `#'`*name* may be used as an abbreviation for `(function` *name*`)`.

function-lambda-expression *Function*

Syntax:

function-lambda-expression *function*
→ *lambda-expression*, *closure-p*, *name*

Arguments and Values:

function—a *function*.

lambda-expression—a *lambda expression* or **nil**.

closure-p—a *generalized boolean*.

name—an *object*.

Description:

Returns information about *function* as follows:

The *primary value*, *lambda-expression*, is *function*'s defining *lambda expression*, or **nil** if the information is not available. The *lambda expression* may have been pre-processed in some ways, but it should remain a suitable argument to **compile** or **function**. Any *implementation* may legitimately return **nil** as the *lambda-expression* of any *function*.

The *secondary value*, *closure-p*, is **nil** if *function*'s definition was enclosed in the *null lexical environment* or something *non-nil* if *function*'s definition might have been enclosed in some *non-null lexical environment*. Any *implementation* may legitimately return *true* as the *closure-p* of any *function*.

The *tertiary value*, *name*, is the "name" of *function*. The name is intended for debugging only and is not necessarily one that would be valid for use as a name in **defun** or **function**, for example. By convention, **nil** is used to mean that *function* has no name. Any *implementation* may legitimately return **nil** as the *name* of any *function*.

Examples:

The following examples illustrate some possible return values, but are not intended to be exhaustive:

```
 (function-lambda-expression #'(lambda (x) x))
→ NIL, false, NIL
or→ NIL, true, NIL
or→ (LAMBDA (X) X), true, NIL
or→ (LAMBDA (X) X), false, NIL

 (function-lambda-expression
    (funcall #'(lambda () #'(lambda (x) x))))
→ NIL, false, NIL
or→ NIL, true, NIL
or→ (LAMBDA (X) X), true, NIL
or→ (LAMBDA (X) X), false, NIL

 (function-lambda-expression
    (funcall #'(lambda (x) #'(lambda () x)) nil))
→ NIL, true, NIL
or→ (LAMBDA () X), true, NIL
not→ NIL, false, NIL
not→ (LAMBDA () X), false, NIL

 (flet ((foo (x) x))
   (setf (symbol-function 'bar) #'foo)
   (function-lambda-expression #'bar))
→ NIL, false, NIL
or→ NIL, true, NIL
or→ (LAMBDA (X) (BLOCK FOO X)), true, NIL
or→ (LAMBDA (X) (BLOCK FOO X)), false, FOO
or→ (SI::BLOCK-LAMBDA FOO (X) X), false, FOO

 (defun foo ()
   (flet ((bar (x) x))
     #'bar))
 (function-lambda-expression (foo))
→ NIL, false, NIL
or→ NIL, true, NIL
or→ (LAMBDA (X) (BLOCK BAR X)), true, NIL
or→ (LAMBDA (X) (BLOCK BAR X)), true, (:INTERNAL FOO 0 BAR)
or→ (LAMBDA (X) (BLOCK BAR X)), false, "BAR in FOO"
```

Notes:

Although *implementations* are free to return "**nil**, *true*, **nil**" in all cases, they are encouraged to return a *lambda expression* as the *primary value* in the case where the argument was created by a call to **compile** or **eval** (as opposed to being created by *loading* a *compiled file*).

functionp *Function*

Syntax:

functionp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of *type* **function**; otherwise, returns *false*.

Examples:

```
(functionp 'append) → false
(functionp #'append) → true
(functionp (symbol-function 'append)) → true
(flet ((f () 1)) (functionp #'f)) → true
(functionp (compile nil '(lambda () 259))) → true
(functionp nil) → false
(functionp 12) → false
(functionp '(lambda (x) (* x x))) → false
(functionp #'(lambda (x) (* x x))) → true
```

Notes:

```
(functionp object) ≡ (typep object 'function)
```

compiled-function-p *Function*

Syntax:

compiled-function-p *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of *type* **compiled-function**; otherwise, returns *false*.

Examples:

```
 (defun f (x) x) → F
 (compiled-function-p #'f)
→ false
or→ true
 (compiled-function-p 'f) → false
 (compile 'f) → F
 (compiled-function-p #'f) → true
 (compiled-function-p 'f) → false
 (compiled-function-p (compile nil '(lambda (x) x)))
→ true
 (compiled-function-p #'(lambda (x) x))
→ false
or→ true
 (compiled-function-p '(lambda (x) x)) → false
```

See Also:

compile, **compile-file**, **compiled-function**

Notes:

```
 (compiled-function-p object) ≡ (typep object 'compiled-function)
```

call-arguments-limit *Constant Variable*

Constant Value:

An integer not smaller than 50 and at least as great as the *value* of **lambda-parameters-limit**, the exact magnitude of which is *implementation-dependent*.

Description:

The upper exclusive bound on the number of *arguments* that may be passed to a *function*.

See Also:

lambda-parameters-limit, **multiple-values-limit**

lambda-list-keywords *Constant Variable*

Constant Value:

a *list*, the *elements* of which are *implementation-dependent*, but which must contain at least the *symbols* **&allow-other-keys**, **&aux**, **&body**, **&environment**, **&key**, **&optional**, **&rest**, and **&whole**.

Description:

A *list* of all the *lambda list keywords* used in the *implementation*, including the additional ones used only by *macro* definition *forms*.

See Also:

defun, **flet**, **defmacro**, **macrolet**, Section 3.1.2 (The Evaluation Model)

lambda-parameters-limit *Constant Variable*

Constant Value:

implementation-dependent, but not smaller than `50`.

Description:

A positive *integer* that is the upper exclusive bound on the number of *parameter names* that can appear in a single *lambda list*.

See Also:

call-arguments-limit

Notes:

Implementors are encouraged to make the *value* of **lambda-parameters-limit** as large as possible.

defconstant *Macro*

Syntax:

defconstant *name initial-value* [*documentation*] → *name*

Arguments and Values:

name—a *symbol*; not evaluated.

initial-value—a *form*; evaluated.

documentation—a *string*; not evaluated.

Description:

defconstant causes the global variable named by *name* to be given a value that is the result of evaluating *initial-value*.

A constant defined by **defconstant** can be redefined with **defconstant**. However, the consequences are undefined if an attempt is made to assign a *value* to the *symbol* using another operator, or to assign it to a *different value* using a subsequent **defconstant**.

If *documentation* is supplied, it is attached to *name* as a *documentation string* of kind **variable**.

defconstant 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 below only take place when **defconstant** appears as a *top level form*.

The consequences are undefined if there are any *bindings* of the variable named by *name* at the time **defconstant** is executed or if the value is not **eql** to the value of *initial-value*.

The consequences are undefined when constant *symbols* are rebound as either lexical or dynamic variables. In other words, a reference to a *symbol* declared with **defconstant** always refers to its global value.

The side effects of the execution of **defconstant** must be equivalent to at least the side effects of the execution of the following code:

```
(setf (symbol-value 'name) initial-value)
(setf (documentation 'name 'variable) 'documentation)
```

If a **defconstant** *form* appears as a *top level form*, the *compiler* must recognize that *name* names a *constant variable*. An implementation may choose to evaluate the value-form at compile time, load time, or both. Therefore, users must ensure that the *initial-value* can be *evaluated* at compile time (regardless of whether or not references to *name* appear in the file) and that it always *evaluates* to the same value.

Examples:

```
 (defconstant this-is-a-constant 'never-changing "for a test") → THIS-IS-A-CONSTANT
this-is-a-constant → NEVER-CHANGING
 (documentation 'this-is-a-constant 'variable) → "for a test"
 (constantp 'this-is-a-constant) → true
```

See Also:

declaim, **defparameter**, **defvar**, **documentation**, **proclaim**, Section 3.1.2.1.1.3 (Constant Variables), Section 3.2 (Compilation)

defparameter, defvar *Macro*

Syntax:

defparameter *name initial-value* [*documentation*] → *name*

defvar *name* [*initial-value* [*documentation*]] → *name*

Arguments and Values:

name—a *symbol*; not evaluated.

initial-value—a *form*; for **defparameter**, it is always *evaluated*, but for **defvar** it is *evaluated* only if *name* is not already *bound*.

documentation—a *string*; not evaluated.

Description:

defparameter and **defvar** *establish* *name* as a *dynamic variable*.

defparameter unconditionally *assigns* the *initial-value* to the *dynamic variable* named *name*. **defvar**, by contrast, *assigns* *initial-value* (if supplied) to the *dynamic variable* named *name* only if *name* is not already *bound*.

If no *initial-value* is supplied, **defvar** leaves the *value cell* of the *dynamic variable* named *name* undisturbed; if *name* was previously *bound*, its old *value* persists, and if it was previously *unbound*, it remains *unbound*.

If *documentation* is supplied, it is attached to *name* as a *documentation string* of kind **variable**.

defparameter and **defvar** normally appear as a *top level form*, but it is meaningful for them to appear as *non-top-level forms*. However, the compile-time side effects described below only take place when they appear as *top level forms*.

Examples:

```
(defparameter *p* 1) → *P*
*p* → 1
(constantp '*p*) → false
(setq *p* 2) → 2
(defparameter *p* 3) → *P*
*p* → 3

(defvar *v* 1) → *V*
*v* → 1
(constantp '*v*) → false
(setq *v* 2) → 2
(defvar *v* 3) → *V*
*v* → 2

(defun foo ()
  (let ((*p* 'p) (*v* 'v))
    (bar))) → FOO
(defun bar () (list *p* *v*)) → BAR
(foo) → (P V)
```

The principal operational distinction between **defparameter** and **defvar** is that **defparameter** makes an unconditional assignment to *name*, while **defvar** makes a conditional one. In practice, this means that **defparameter** is useful in situations where loading or reloading the definition would want to pick up a new value of the variable, while **defvar** is used in situations where the old value would want to be retained if the file were loaded or reloaded. For example, one might create a file which contained:

```
(defvar *the-interesting-numbers* '())
(defmacro define-interesting-number (name n)
  `(progn (defvar ,name ,n)
          (pushnew ,name *the-interesting-numbers*)
          ',name))
(define-interesting-number *my-height* 168) ;cm
(define-interesting-number *my-weight* 13)  ;stones
```

Here the initial value, `()`, for the variable `*the-interesting-numbers*` is just a seed that we are never likely to want to reset to something else once something has been grown from it. As such, we have used **defvar** to avoid having the `*interesting-numbers*` information reset if the file is loaded a second time. It is true that the two calls to **define-interesting-number** here would be reprocessed, but if there were additional calls in another file, they would not be and that information would be lost. On the other hand, consider the following code:

```
(defparameter *default-beep-count* 3)
(defun beep (&optional (n *default-beep-count*))
  (dotimes (i n) (si:%beep 1000. 100000.) (sleep 0.1)))
```

defparameter, defvar

Here we could easily imagine editing the code to change the initial value of `*default-beep-count*`, and then reloading the file to pick up the new value. In order to make value updating easy, we have used **defparameter**.

On the other hand, there is potential value to using **defvar** in this situation. For example, suppose that someone had predefined an alternate value for `*default-beep-count*`, or had loaded the file and then manually changed the value. In both cases, if we had used **defvar** instead of **defparameter**, those user preferences would not be overridden by (re)loading the file.

The choice of whether to use **defparameter** or **defvar** has visible consequences to programs, but is nevertheless often made for subjective reasons.

Side Effects:

If a **defvar** or **defparameter** *form* appears as a *top level form*, the *compiler* must recognize that the *name* has been proclaimed **special**. However, it must neither *evaluate* the *initial-value form* nor *assign* the *dynamic variable* named *name* at compile time.

There may be additional (*implementation-defined*) compile-time or run-time side effects, as long as such effects do not interfere with the correct operation of *conforming programs*.

Affected By:

defvar is affected by whether *name* is already *bound*.

See Also:

declaim, **defconstant**, **documentation**, Section 3.2 (Compilation)

Notes:

It is customary to name *dynamic variables* with an *asterisk* at the beginning and end of the name. e.g., `*foo*` is a good name for a *dynamic variable*, but not for a *lexical variable*; `foo` is a good name for a *lexical variable*, but not for a *dynamic variable*. This naming convention is observed for all *defined names* in Common Lisp; however, neither *conforming programs* nor *conforming implementations* are obliged to adhere to this convention.

The intent of the permission for additional side effects is to allow *implementations* to do normal "bookkeeping" that accompanies definitions. For example, the *macro expansion* of a **defvar** or **defparameter** *form* might include code that arranges to record the name of the source file in which the definition occurs.

defparameter and **defvar** might be defined as follows:

```
(defmacro defparameter (name initial-value
                        &optional (documentation nil documentation-p))
  `(progn (declaim (special ,name))
          (setf (symbol-value ',name) ,initial-value)
          ,(when documentation-p
             `(setf (documentation ',name 'variable) ',documentation))
          ',name))
```

```
(defmacro defvar (name &optional
                       (initial-value nil initial-value-p)
                       (documentation nil documentation-p))
  `(progn (declaim (special ,name))
          ,(when initial-value-p
             `(unless (boundp ',name)
                (setf (symbol-value ',name) ,initial-value)))
          ,(when documentation-p
             `(setf (documentation ',name 'variable) ',documentation))
          ',name))
```

destructuring-bind *Macro*

Syntax:

destructuring-bind *lambda-list expression* {*declaration*}* {*form*}*
→ {*result*}*

Arguments and Values:

lambda-list—a *destructuring lambda list*.

expression—a *form*.

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

forms—an *implicit progn*.

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

Description:

destructuring-bind binds the variables specified in *lambda-list* to the corresponding values in the tree structure resulting from the evaluation of *expression*; then **destructuring-bind** evaluates *forms*.

The *lambda-list* supports destructuring as described in Section 3.4.5 (Destructuring Lambda Lists).

Examples:

```
(defun iota (n) (loop for i from 1 to n collect i))       ;helper
(destructuring-bind ((a &optional (b 'bee)) one two three)
    `((alpha) ,@(iota 3))
  (list a b three two one)) → (ALPHA BEE 3 2 1)
```

Exceptional Situations:

If the result of evaluating the *expression* does not match the destructuring pattern, an error of *type* **error** should be signaled.

See Also:

macrolet, **defmacro**

let, let∗ *Special Operator*

Syntax:

let ({*var* | (*var* [*init-form*])}*) {*declaration*}* {*form*}* → {*result*}*

let* ({*var* | (*var* [*init-form*])}*) {*declaration*}* {*form*}* → {*result*}*

Arguments and Values:

var—a *symbol*.

init-form—a *form*.

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

form—a *form*.

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

Description:

let and **let*** create new variable *bindings* and execute a series of *forms* that use these *bindings*. **let** performs the *bindings* in parallel and **let*** does them sequentially.

The form

```
(let ((var1 init-form-1)
      (var2 init-form-2)
      ...
      (varm init-form-m))
  declaration1
  declaration2
  ...
  declarationp
  form1
  form2
  ...
  formn)
```

first evaluates the expressions *init-form-1*, *init-form-2*, and so on, in that order, saving the resulting values. Then all of the variables *varj* are bound to the corresponding values; each *binding* is lexical unless there is a **special** declaration to the contrary. The expressions *formk* are then evaluated in order; the values of all but the last are discarded (that is, the body of a **let** is an *implicit progn*).

let* is similar to **let**, but the *bindings* of variables are performed sequentially rather than in parallel. The expression for the *init-form* of a *var* can refer to *vars* previously bound in the **let***.

The form

```
(let* ((var1 init-form-1)
       (var2 init-form-2)
       ...
       (varm init-form-m))
  declaration1
  declaration2
  ...
  declarationp
  form1
  form2
  ...
  formn)
```

first evaluates the expression *init-form-1*, then binds the variable *var1* to that value; then it evaluates *init-form-2* and binds *var2*, and so on. The expressions *formj* are then evaluated in order; the values of all but the last are discarded (that is, the body of **let*** is an implicit **progn**).

For both **let** and **let***, if there is not an *init-form* associated with a *var*, *var* is initialized to **nil**.

The special form **let** has the property that the *scope* of the name binding does not include any initial value form. For **let***, a variable's *scope* also includes the remaining initial value forms for subsequent variable bindings.

Examples:

```
(setq a 'top) → TOP
(defun dummy-function () a) → DUMMY-FUNCTION
(let ((a 'inside) (b a))
   (format nil "~S ~S ~S" a b (dummy-function))) → "INSIDE TOP TOP"
(let* ((a 'inside) (b a))
   (format nil "~S ~S ~S" a b (dummy-function))) → "INSIDE INSIDE TOP"
(let ((a 'inside) (b a))
   (declare (special a))
   (format nil "~S ~S ~S" a b (dummy-function))) → "INSIDE TOP INSIDE"
```

The code

```
 (let (x)
   (declare (integer x))
   (setq x (gcd y z))
   ...)
```

is incorrect; although `x` is indeed set before it is used, and is set to a value of the declared type *integer*, nevertheless `x` initially takes on the value **nil** in violation of the type declaration.

See Also:

progv

progv *Special Operator*

Syntax:

progv *symbols values* {*form*}* → {*result*}*

Arguments and Values:

symbols—a *list* of *symbols*; evaluated.

values—a *list* of *objects*; evaluated.

forms—an *implicit progn*.

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

Description:

progv creates new dynamic variable *bindings* and executes each *form* using those *bindings*. Each *form* is evaluated in order.

progv allows *binding* one or more dynamic variables whose names may be determined at run time. Each *form* is evaluated in order with the dynamic variables whose names are in *symbols* bound to corresponding *values*. If too few *values* are supplied, the remaining *symbols* are bound and then made to have no value. If too many *values* are supplied, the excess values are ignored. The *bindings* of the dynamic variables are undone on exit from **progv**.

Examples:

```
 (setq *x* 1) → 1
 (progv '(*x*) '(2) *x*) → 2
 *x* → 1

Assuming *x* is not globally special,

 (let ((*x* 3))
    (progv '(*x*) '(4)
      (list *x* (symbol-value '*x*)))) → (3 4)
```

See Also:

let, Section 3.1 (Evaluation)

Notes:

Among other things, **progv** is useful when writing interpreters for languages embedded in Lisp; it provides a handle on the mechanism for *binding dynamic variables*.

setq *Special Form*

Syntax:

setq {↓*pair*}* → *result*

pair::=*var form*

Pronunciation:

[ˈ**set**ˌ**kyü**]

Arguments and Values:

var—a *symbol* naming a *variable* other than a *constant variable*.

form—a *form*.

result—the *primary value* of the last *form*, or **nil** if no *pairs* were supplied.

Description:

Assigns values to *variables*.

(setq *var1 form1 var2 form2* ...) is the simple variable assignment statement of Lisp. First *form1* is evaluated and the result is stored in the variable *var1*, then *form2* is evaluated and the result stored in *var2*, and so forth. **setq** may be used for assignment of both lexical and dynamic variables.

If any *var* refers to a *binding* made by **symbol-macrolet**, then that *var* is treated as if **setf** (not **setq**) had been used.

Examples:

```
;; A simple use of SETQ to establish values for variables.
(setq a 1 b 2 c 3) → 3
a → 1
b → 2
c → 3
```

```
 ;; Use of SETQ to update values by sequential assignment.
 (setq a (1+ b) b (1+ a) c (+ a b)) → 7
 a → 3
 b → 4
 c → 7

 ;; This illustrates the use of SETQ on a symbol macro.
 (let ((x (list 10 20 30)))
   (symbol-macrolet ((y (car x)) (z (cadr x)))
     (setq y (1+ z) z (1+ y))
     (list x y z)))
→ ((21 22 30) 21 22)
```

Side Effects:

The *primary value* of each *form* is assigned to the corresponding *var*.

See Also:

psetq, **set**, **setf**

psetq *Macro*

Syntax:

psetq {↓*pair*}* → **nil**

pair::=*var form*

Pronunciation:

psetq: [ˈpē ˈset ˌkyü]

Arguments and Values:

var—a *symbol* naming a *variable* other than a *constant variable*.

form—a *form*.

Description:

Assigns values to *variables*.

This is just like **setq**, except that the assignments happen "in parallel." That is, first all of the forms are evaluated, and only then are the variables set to the resulting values. In this way, the assignment to one variable does not affect the value computation of another in the way that would occur with **setq**'s sequential assignment.

If any *var* refers to a *binding* made by **symbol-macrolet**, then that *var* is treated as if **psetf** (not **psetq**) had been used.

Examples:

```
;; A simple use of PSETQ to establish values for variables.
;; As a matter of style, many programmers would prefer SETQ
;; in a simple situation like this where parallel assignment
;; is not needed, but the two have equivalent effect.
(psetq a 1 b 2 c 3) → NIL
a → 1
b → 2
c → 3

;; Use of PSETQ to update values by parallel assignment.
;; The effect here is very different than if SETQ had been used.
(psetq a (1+ b) b (1+ a) c (+ a b)) → NIL
a → 3
b → 2
c → 3

;; Use of PSETQ on a symbol macro.
(let ((x (list 10 20 30)))
  (symbol-macrolet ((y (car x)) (z (cadr x)))
    (psetq y (1+ z) z (1+ y))
    (list x y z)))
→ ((21 11 30) 21 11)

;; Use of parallel assignment to swap values of A and B.
(let ((a 1) (b 2))
  (psetq a b  b a)
  (values a b))
→ 2, 1
```

Side Effects:

The values of *forms* are assigned to *vars*.

See Also:

psetf, **setq**

block *Special Operator*

Syntax:

block *name form** → {*result*}*

Arguments and Values:

name—a *symbol*.

form—a *form*.

results—the *values* of the *forms* if a *normal return* occurs, or else, if an *explicit return* occurs, the *values* that were transferred.

Description:

block *establishes* a *block* named *name* and then evaluates *forms* as an *implicit progn*.

The *special operators* **block** and **return-from** work together to provide a structured, lexical, non-local exit facility. At any point lexically contained within *forms*, **return-from** can be used with the given *name* to return control and values from the **block** *form*, except when an intervening *block* with the same name has been *established*, in which case the outer *block* is shadowed by the inner one.

The *block* named *name* has *lexical scope* and *dynamic extent*.

Once established, a *block* may only be exited once, whether by *normal return* or *explicit return*.

Examples:

```
(block empty) → NIL
(block whocares (values 1 2) (values 3 4)) → 3, 4
(let ((x 1))
  (block stop (setq x 2) (return-from stop) (setq x 3))
  x) → 2
(block early (return-from early (values 1 2)) (values 3 4)) → 1, 2
(block outer (block inner (return-from outer 1)) 2) → 1
(block twin (block twin (return-from twin 1)) 2) → 2
;; Contrast behavior of this example with corresponding example of CATCH.
(block b
  (flet ((b1 () (return-from b 1)))
    (block b (b1) (print 'unreachable))
    2)) → 1
```

See Also:

return, **return-from**, Section 3.1 (Evaluation)

Notes:

catch *Special Operator*

Syntax:

catch *tag* {*form*}* → {*result*}*

Arguments and Values:

tag—a *catch tag*; evaluated.

forms—an *implicit progn*.

results—if the *forms* exit normally, the *values* returned by the *forms*; if a throw occurs to the *tag*, the *values* that are thrown.

Description:

catch is used as the destination of a non-local control transfer by **throw**. *Tags* are used to find the **catch** to which a **throw** is transferring control. `(catch 'foo` *form*`)` catches a `(throw 'foo` *form*`)` but not a `(throw 'bar` *form*`)`.

The order of execution of **catch** follows:

1. *Tag* is evaluated. It serves as the name of the **catch**.

2. *Forms* are then evaluated as an implicit **progn**, and the results of the last *form* are returned unless a **throw** occurs.

3. If a **throw** occurs during the execution of one of the *forms*, control is transferred to the **catch** *form* whose *tag* is **eq** to the tag argument of the **throw** and which is the most recently established **catch** with that *tag*. No further evaluation of *forms* occurs.

4. The *tag* *established* by **catch** is *disestablished* just before the results are returned.

If during the execution of one of the *forms*, a **throw** is executed whose tag is **eq** to the **catch** tag, then the values specified by the **throw** are returned as the result of the dynamically most recently established **catch** form with that tag.

The mechanism for **catch** and **throw** works even if **throw** is not within the lexical scope of **catch**. **throw** must occur within the *dynamic extent* of the *evaluation* of the body of a **catch** with a corresponding *tag*.

Examples:

```
(catch 'dummy-tag 1 2 (throw 'dummy-tag 3) 4) → 3
```

```
(catch 'dummy-tag 1 2 3 4) → 4
(defun throw-back (tag) (throw tag t)) → THROW-BACK
(catch 'dummy-tag (throw-back 'dummy-tag) 2) → T

;; Contrast behavior of this example with corresponding example of BLOCK.
(catch 'c
  (flet ((c1 () (throw 'c 1)))
    (catch 'c (c1) (print 'unreachable))
    2)) → 2
```

Exceptional Situations:

An error of *type* **control-error** is signaled if **throw** is done when there is no suitable **catch** *tag*.

See Also:

throw, Section 3.1 (Evaluation)

Notes:

It is customary for *symbols* to be used as *tags*, but any *object* is permitted. However, numbers should not be used because the comparison is done using **eq**.

catch differs from **block** in that **catch** tags have dynamic *scope* while **block** names have *lexical scope*.

go *Special Operator*

Syntax:

go *tag* →|

Arguments and Values:

tag—a *go tag*.

Description:

go transfers control to the point in the body of an enclosing **tagbody** form labeled by a tag **eql** to *tag*. If there is no such *tag* in the body, the bodies of lexically containing **tagbody** *forms* (if any) are examined as well. If several tags are **eql** to *tag*, control is transferred to whichever matching *tag* is contained in the innermost **tagbody** form that contains the **go**. The consequences are undefined if there is no matching *tag* lexically visible to the point of the **go**.

The transfer of control initiated by **go** is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

Examples:

```
(tagbody
   (setq val 2)
   (go lp)
   (incf val 3)
   lp (incf val 4)) → NIL
 val → 6
```

The following is in error because there is a normal exit of the **tagbody** before the **go** is executed.

```
 (let ((a nil))
   (tagbody t (setq a #'(lambda () (go t))))
   (funcall a))
```

The following is in error because the **tagbody** is passed over before the **go** *form* is executed.

```
 (funcall (block nil
            (tagbody a (return #'(lambda () (go a))))))
```

See Also:

tagbody

return-from *Special Operator*

Syntax:

return-from *name* [*result*] →|

Arguments and Values:

name—a *block tag*; not evaluated.

result—a *form*; evaluated. The default is **nil**.

Description:

Returns control and *multiple values*$_2$ from a lexically enclosing *block*.

A **block** *form* named *name* must lexically enclose the occurrence of **return-from**; any *values yielded* by the *evaluation* of *result* are immediately returned from the innermost such lexically enclosing *block*.

The transfer of control initiated by **return-from** is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

Examples:

```
 (block alpha (return-from alpha) 1) → NIL
 (block alpha (return-from alpha 1) 2) → 1
```

return-from

```
 (block alpha (return-from alpha (values 1 2)) 3) → 1, 2
 (let ((a 0))
    (dotimes (i 10) (incf a) (when (oddp i) (return)))
    a) → 2
 (defun temp (x)
    (if x (return-from temp 'dummy))
    44) → TEMP
 (temp nil) → 44
 (temp t) → DUMMY
 (block out
   (flet ((exit (n) (return-from out n)))
     (block out (exit 1)))
   2) → 1
 (block nil
   (unwind-protect (return-from nil 1)
     (return-from nil 2)))
→ 2
 (dolist (flag '(nil t))
   (block nil
     (let ((x 5))
       (declare (special x))
       (unwind-protect (return-from nil)
         (print x))))
   (print 'here))
▷ 5
▷ HERE
▷ 5
▷ HERE
→ NIL
 (dolist (flag '(nil t))
   (block nil
     (let ((x 5))
       (declare (special x))
       (unwind-protect
           (if flag (return-from nil))
         (print x))))
   (print 'here))
▷ 5
▷ HERE
▷ 5
▷ HERE
→ NIL
```

The following has undefined consequences because the **block** *form* exits normally before the **return-from** *form* is attempted.

```
(funcall (block nil #'(lambda () (return-from nil)))) is an error.
```

See Also:

block, **return**, Section 3.1 (Evaluation)

return *Macro*

Syntax:

return [*result*] $\rightarrow|$

Arguments and Values:

result—a *form*; evaluated. The default is **nil**.

Description:

Returns, as if by **return-from**, from the *block* named **nil**.

Examples:

```
(block nil (return) 1) → NIL
(block nil (return 1) 2) → 1
(block nil (return (values 1 2)) 3) → 1, 2
(block nil (block alpha (return 1) 2)) → 1
(block alpha (block nil (return 1)) 2) → 2
(block nil (block nil (return 1) 2)) → 1
```

See Also:

block, **return-from**, Section 3.1 (Evaluation)

Notes:

```
(return) ≡ (return-from nil)
(return form) ≡ (return-from nil form)
```

The *implicit blocks established* by *macros* such as **do** are often named **nil**, so that **return** can be used to exit from such *forms*.

tagbody

tagbody *Special Operator*

Syntax:

tagbody {*tag* | *statement*}* → **nil**

Arguments and Values:

tag—a *go tag*; not evaluated.

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

Description:

Executes zero or more *statements* in a *lexical environment* that provides for control transfers to labels indicated by the *tags*.

The *statements* in a **tagbody** are *evaluated* in order from left to right, and their *values* are discarded. If at any time there are no remaining *statements*, **tagbody** returns **nil**. However, if (go *tag*) is *evaluated*, control jumps to the part of the body labeled with the *tag*. (Tags are compared with **eql**.)

A *tag* established by **tagbody** has *lexical scope* and has *dynamic extent*. Once **tagbody** has been exited, it is no longer valid to **go** to a *tag* in its body. It is permissible for **go** to jump to a **tagbody** that is not the innermost **tagbody** containing that **go**; the *tags* established by a **tagbody** only shadow other *tags* of like name.

The determination of which elements of the body are *tags* and which are *statements* is made prior to any *macro expansion* of that element. If a *statement* is a *macro form* and its *macro expansion* is an *atom*, that *atom* is treated as a *statement*, not a *tag*.

Examples:

```
(let (val)
   (tagbody
     (setq val 1)
     (go point-a)
     (incf val 16)
    point-c
     (incf val 04)
     (go point-b)
     (incf val 32)
    point-a
     (incf val 02)
     (go point-c)
     (incf val 64)
    point-b
     (incf val 08))
    val)
→ 15
 (defun f1 (flag)
   (let ((n 1))
     (tagbody
       (setq n (f2 flag #'(lambda () (go out))))
      out
       (prin1 n))))
→ F1
 (defun f2 (flag escape)
   (if flag (funcall escape) 2))
→ F2
 (f1 nil)
▷ 2
→ NIL
 (f1 t)
▷ 1
→ NIL
```

See Also:

go

Notes:

The *macros* in Figure 5–10 have *implicit tagbodies.*

do	**do-external-symbols**	**dotimes**
do*	**do-symbols**	**prog**
do-all-symbols	**dolist**	**prog***

Figure 5–10. Macros that have implicit tagbodies.

throw *Special Operator*

Syntax:

throw *tag result-form* →|

Arguments and Values:

tag—a *catch tag*; evaluated.

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

throw

Description:

throw causes a non-local control transfer to a **catch** whose tag is **eq** to *tag*.

Tag is evaluated first to produce an *object* called the throw tag; then *result-form* is evaluated, and its results are saved. If the *result-form* produces multiple values, then all the values are saved. The most recent outstanding **catch** whose *tag* is **eq** to the throw tag is exited; the saved results are returned as the value or values of **catch**.

The transfer of control initiated by **throw** is performed as described in Section 5.2 (Transfer of Control to an Exit Point).

Examples:

```
(catch 'result
   (setq i 0 j 0)
   (loop (incf j 3) (incf i)
         (if (= i 3) (throw 'result (values i j))))) → 3, 9
```

```
(catch nil
  (unwind-protect (throw nil 1)
    (throw nil 2))) → 2
```

The consequences of the following are undefined because the **catch** of `b` is passed over by the first **throw**, hence portable programs must assume that its *dynamic extent* is terminated. The *binding* of the *catch tag* is not yet *disestablished* and therefore it is the target of the second **throw**.

```
(catch 'a
  (catch 'b
    (unwind-protect (throw 'a 1)
      (throw 'b 2))))
```

The following prints "`The inner catch returns :SECOND-THROW`" and then returns `:outer-catch`.

```
(catch 'foo
        (format t "The inner catch returns ~s.~%"
                (catch 'foo
                    (unwind-protect (throw 'foo :first-throw)
                        (throw 'foo :second-throw))))
        :outer-catch)
▷ The inner catch returns :SECOND-THROW
→ :OUTER-CATCH
```

Exceptional Situations:

If there is no outstanding *catch tag* that matches the throw tag, no unwinding of the stack is performed, and an error of *type* **control-error** is signaled. When the error is signaled, the *dynamic environment* is that which was in force at the point of the **throw**.

See Also:

block, **catch**, **return-from**, **unwind-protect**, Section 3.1 (Evaluation)

Notes:

catch and **throw** are normally used when the *exit point* must have *dynamic scope* (*e.g.*, the **throw** is not lexically enclosed by the **catch**), while **block** and **return** are used when *lexical scope* is sufficient.

unwind-protect *Special Operator*

Syntax:

unwind-protect *protected-form* {*cleanup-form*}* → {*result*}*

Arguments and Values:

protected-form—a *form*.

cleanup-form—a *form*.

results—the *values* of the *protected-form*.

Description:

unwind-protect evaluates *protected-form* and guarantees that *cleanup-forms* are executed before **unwind-protect** exits, whether it terminates normally or is aborted by a control transfer of some kind. **unwind-protect** is intended to be used to make sure that certain side effects take place after the evaluation of *protected-form*.

If a *non-local exit* occurs during execution of *cleanup-forms*, no special action is taken. The *cleanup-forms* of **unwind-protect** are not protected by that **unwind-protect**.

unwind-protect protects against all attempts to exit from *protected-form*, including **go**, **handler-case**, **ignore-errors**, **restart-case**, **return-from**, **throw**, and **with-simple-restart**.

Undoing of *handler* and *restart bindings* during an exit happens in parallel with the undoing of the bindings of *dynamic variables* and **catch** tags, in the reverse order in which they were established. The effect of this is that *cleanup-form* sees the same *handler* and *restart bindings*, as well as *dynamic variable bindings* and **catch** tags, as were visible when the **unwind-protect** was entered.

Examples:

```
 (tagbody
```

unwind-protect

```
  (let ((x 3))
    (unwind-protect
      (if (numberp x) (go out))
      (print x)))
 out
  ...)
```

When **go** is executed, the call to **print** is executed first, and then the transfer of control to the tag `out` is completed.

```
 (defun dummy-function (x)
    (setq state 'running)
    (unless (numberp x) (throw 'abort 'not-a-number))
    (setq state (1+ x))) → DUMMY-FUNCTION
 (catch 'abort (dummy-function 1)) → 2
 state → 2
 (catch 'abort (dummy-function 'trash)) → NOT-A-NUMBER
 state → RUNNING
 (catch 'abort (unwind-protect (dummy-function 'trash)
                  (setq state 'aborted))) → NOT-A-NUMBER
 state → ABORTED
```

The following code is not correct:

```
 (unwind-protect
   (progn (incf *access-count*)
          (perform-access))
   (decf *access-count*))
```

If an exit occurs before completion of **incf**, the **decf** *form* is executed anyway, resulting in an incorrect value for `*access-count*`. The correct way to code this is as follows:

```
 (let ((old-count *access-count*))
   (unwind-protect
     (progn (incf *access-count*)
            (perform-access))
     (setq *access-count* old-count)))


;;; The following returns 2.
 (block nil
   (unwind-protect (return 1)
     (return 2)))

;;; The following has undefined consequences.
 (block a
   (block b
     (unwind-protect (return-from a 1)
       (return-from b 2))))

;;; The following returns 2.
 (catch nil
   (unwind-protect (throw nil 1)
     (throw nil 2)))

;;; The following has undefined consequences because the catch of B is
;;; passed over by the first THROW, hence portable programs must assume
;;; its dynamic extent is terminated.  The binding of the catch tag is not
;;; yet disestablished and therefore it is the target of the second throw.
 (catch 'a
   (catch 'b
     (unwind-protect (throw 'a 1)
       (throw 'b 2))))

;;; The following prints "The inner catch returns :SECOND-THROW"
;;; and then returns :OUTER-CATCH.
 (catch 'foo
         (format t "The inner catch returns ~s.~%"
                 (catch 'foo
                     (unwind-protect (throw 'foo :first-throw)
                         (throw 'foo :second-throw))))
         :outer-catch)


;;; The following returns 10. The inner CATCH of A is passed over, but
;;; because that CATCH is disestablished before the THROW to A is executed,
;;; it isn't seen.
 (catch 'a
   (catch 'b
     (unwind-protect (1+ (catch 'a (throw 'b 1)))
       (throw 'a 10))))


;;; The following has undefined consequences because the extent of
;;; the (CATCH 'BAR ...) exit ends when the (THROW 'FOO ...)
;;; commences.
 (catch 'foo
   (catch 'bar
       (unwind-protect (throw 'foo 3)
         (throw 'bar 4)
         (print 'xxx))))
```

```
;;; The following returns 4; XXX is not printed.
;;; The (THROW 'FOO ...) has no effect on the scope of the BAR
;;; catch tag or the extent of the (CATCH 'BAR ...) exit.
 (catch 'bar
   (catch 'foo
       (unwind-protect (throw 'foo 3)
         (throw 'bar 4)
         (print 'xxx))))


;;; The following prints 5.
 (block nil
   (let ((x 5))
     (declare (special x))
     (unwind-protect (return)
       (print x))))
```

See Also:

catch, **go**, **handler-case**, **restart-case**, **return**, **return-from**, **throw**, Section 3.1 (Evaluation)

nil *Constant Variable*

Constant Value:

nil.

Description:

nil represents both *boolean* (and *generalized boolean*) *false* and the *empty list*.

Examples:

```
 nil → NIL
```

See Also:

t

not *Function*

Syntax:

not *x* → *boolean*

Arguments and Values:

x—a *generalized boolean* (*i.e.*, any *object*).

boolean—a *boolean*.

Description:

Returns **t** if *x* is *false*; otherwise, returns **nil**.

Examples:

```
(not nil) → T
(not '()) → T
(not (integerp 'sss)) → T
(not (integerp 1)) → NIL
(not 3.7) → NIL
(not 'apple) → NIL
```

See Also:

null

Notes:

not is intended to be used to invert the 'truth value' of a *boolean* (or *generalized boolean*) whereas **null** is intended to be used to test for the *empty list*. Operationally, **not** and **null** compute the same result; which to use is a matter of style.

t *Constant Variable*

Constant Value:

t.

Description:

The *boolean* representing true, and the canonical *generalized boolean* representing true. Although any *object* other than **nil** is considered *true*, **t** is generally used when there is no special reason to prefer one such *object* over another.

The *symbol* **t** is also sometimes used for other purposes as well. For example, as the *name* of a *class*, as a *designator* (*e.g.*, a *stream designator*) or as a special symbol for some syntactic reason (*e.g.*, in **case** and **typecase** to label the *otherwise-clause*).

Examples:

```
 t → T
 (eq t 't) → true
 (find-class 't) → #<CLASS T 610703333>
 (case 'a (a 1) (t 2)) → 1
 (case 'b (a 1) (t 2)) → 2
 (prin1 'hello t)
▷ HELLO
→ HELLO
```

See Also:

nil

eq *Function*

Syntax:

eq *x y* → *generalized-boolean*

Arguments and Values:

x—an *object*.

y—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if its *arguments* are the same, identical *object*; otherwise, returns *false*.

Examples:

```
 (eq 'a 'b) → false
 (eq 'a 'a) → true
 (eq 3 3)
→ true
or→ false
 (eq 3 3.0) → false
 (eq 3.0 3.0)
→ true
or→ false
```

```
 (eq #c(3 -4) #c(3 -4))
→ true
or
→ false
 (eq #c(3 -4.0) #c(3 -4)) → false
 (eq (cons 'a 'b) (cons 'a 'c)) → false
 (eq (cons 'a 'b) (cons 'a 'b)) → false
 (eq '(a . b) '(a . b))
→ true
or
→ false
 (progn (setq x (cons 'a 'b)) (eq x x)) → true
 (progn (setq x '(a . b)) (eq x x)) → true
 (eq #\A #\A)
→ true
or
→ false
 (let ((x "Foo")) (eq x x)) → true
 (eq "Foo" "Foo")
→ true
or
→ false
 (eq "Foo" (copy-seq "Foo")) → false
 (eq "FOO" "foo") → false
 (eq "string-seq" (copy-seq "string-seq")) → false
 (let ((x 5)) (eq x x))
→ true
or
→ false
```

See Also:

eql, **equal**, **equalp**, **=**, Section 3.2 (Compilation)

Notes:

Objects that appear the same when printed are not necessarily **eq** to each other. *Symbols* that print the same usually are **eq** to each other because of the use of the **intern** function. However, *numbers* with the same value need not be **eq**, and two similar *lists* are usually not *identical*.

An implementation is permitted to make "copies" of *characters* and *numbers* at any time. The effect is that Common Lisp makes no guarantee that **eq** is true even when both its arguments are "the same thing" if that thing is a *character* or *number*.

Most Common Lisp *operators* use **eql** rather than **eq** to compare objects, or else they default to **eql** and only use **eq** if specifically requested to do so. However, the following *operators* are defined to use **eq** rather than **eql** in a way that cannot be overridden by the *code* which employs them:

catch	**getf**	**throw**
get	**remf**	
get-properties	**remprop**	

Figure 5–11. Operators that always prefer EQ over EQL

eql *Function*

Syntax:

eql *x y* → *generalized-boolean*

Arguments and Values:

x—an *object*.

y—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

The value of **eql** is *true* of two objects, *x* and *y*, in the folowing cases:

1. If *x* and *y* are **eq**.
2. If *x* and *y* are both *numbers* of the same *type* and the same value.
3. If they are both *characters* that represent the same character.

Otherwise the value of **eql** is *false*.

If an implementation supports positive and negative zeros as *distinct* values, then `(eql 0.0 -0.0)` returns *false*. Otherwise, when the syntax `-0.0` is read it is interpreted as the value `0.0`, and so `(eql 0.0 -0.0)` returns *true*.

Examples:

```
 (eql 'a 'b) → false
 (eql 'a 'a) → true
 (eql 3 3) → true
 (eql 3 3.0) → false
 (eql 3.0 3.0) → true
 (eql #c(3 -4) #c(3 -4)) → true
 (eql #c(3 -4.0) #c(3 -4)) → false
 (eql (cons 'a 'b) (cons 'a 'c)) → false
 (eql (cons 'a 'b) (cons 'a 'b)) → false
 (eql '(a . b) '(a . b))
→ true
or→ false
 (progn (setq x (cons 'a 'b)) (eql x x)) → true
 (progn (setq x '(a . b)) (eql x x)) → true
 (eql #\A #\A) → true
 (eql "Foo" "Foo")
→ true
or→ false
 (eql "Foo" (copy-seq "Foo")) → false
 (eql "FOO" "foo") → false
```

Normally (eql 1.0s0 1.0d0) is false, under the assumption that 1.0s0 and 1.0d0 are of distinct data types. However, implementations that do not provide four distinct floating-point formats are permitted to "collapse" the four formats into some smaller number of them; in such an implementation (eql 1.0s0 1.0d0) might be true.

See Also:

eq, **equal**, **equalp**, **=**, **char=**

Notes:

eql is the same as **eq**, except that if the arguments are *characters* or *numbers* of the same type then their values are compared. Thus **eql** tells whether two *objects* are conceptually the same, whereas **eq** tells whether two *objects* are implementationally identical. It is for this reason that **eql**, not **eq**, is the default comparison predicate for *operators* that take *sequences* as arguments.

eql may not be true of two *floats* even when they represent the same value. **=** is used to compare mathematical values.

Two *complex* numbers are considered to be **eql** if their real parts are **eql** and their imaginary parts are **eql**. For example, (eql #C(4 5) #C(4 5)) is *true* and (eql #C(4 5) #C(4.0 5.0)) is *false*. Note that while (eql #C(5.0 0.0) 5.0) is *false*, (eql #C(5 0) 5) is *true*. In the case of (eql #C(5.0 0.0) 5.0) the two arguments are of different types, and so cannot satisfy **eql**. In the case of (eql #C(5 0) 5), #C(5 0) is not a *complex* number, but is automatically reduced to the *integer* 5.

equal *Function*

Syntax:

equal *x y* → *generalized-boolean*

Arguments and Values:

x—an *object*.

y—an *object*.

generalized-boolean—a *generalized boolean*.

equal

Description:

Returns *true* if *x* and *y* are structurally similar (isomorphic) *objects*. *Objects* are treated as follows by **equal**.

Symbols, *Numbers*, and *Characters*

> **equal** is *true* of two *objects* if they are *symbols* that are **eq**, if they are *numbers* that are **eql**, or if they are *characters* that are **eql**.

Conses

> For *conses*, **equal** is defined recursively as the two *cars* being **equal** and the two *cdrs* being **equal**.

Arrays

> Two *arrays* are **equal** only if they are **eq**, with one exception: *strings* and *bit vectors* are compared element-by-element (using **eql**). If either *x* or *y* has a *fill pointer*, the *fill pointer* limits the number of elements examined by **equal**. Uppercase and lowercase letters in *strings* are considered by **equal** to be different.

Pathnames

> Two *pathnames* are **equal** if and only if all the corresponding components (host, device, and so on) are equivalent. Whether or not uppercase and lowercase letters are considered equivalent in *strings* appearing in components is *implementation-dependent*. *pathnames* that are **equal** should be functionally equivalent.

Other (Structures, hash-tables, instances, ...)

> Two other *objects* are **equal** only if they are **eq**.

equal does not descend any *objects* other than the ones explicitly specified above. Figure 5–12 summarizes the information given in the previous list. In addition, the figure specifies the priority of the behavior of **equal**, with upper entries taking priority over lower ones.

Type	**Behavior**
number	uses **eql**
character	uses **eql**
cons	descends
bit vector	descends
string	descends
pathname	"functionally equivalent"
structure	uses **eq**
Other *array*	uses **eq**
hash table	uses **eq**
Other *object*	uses **eq**

Figure 5–12. Summary and priorities of behavior of equal

Any two *objects* that are **eql** are also **equal**.

equal may fail to terminate if *x* or *y* is circular.

Examples:

```
(equal 'a 'b) → false
(equal 'a 'a) → true
(equal 3 3) → true
(equal 3 3.0) → false
(equal 3.0 3.0) → true
(equal #c(3 -4) #c(3 -4)) → true
(equal #c(3 -4.0) #c(3 -4)) → false
(equal (cons 'a 'b) (cons 'a 'c)) → false
(equal (cons 'a 'b) (cons 'a 'b)) → true
(equal #\A #\A) → true
(equal #\A #\a) → false
(equal "Foo" "Foo") → true
(equal "Foo" (copy-seq "Foo")) → true
(equal "FOO" "foo") → false
(equal "This-string" "This-string") → true
(equal "This-string" "this-string") → false
```

See Also:

eq, **eql**, **equalp**, **=**, **string=**, **string-equal**, **char=**, **char-equal**, **tree-equal**

Notes:

Object equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound very generic, **equal** and **equalp** are not appropriate for every application.

A rough rule of thumb is that two *objects* are **equal** if and only if their printed representations are the same.

equalp *Function*

Syntax:

equalp *x y* → *generalized-boolean*

Arguments and Values:

x—an *object*.

y—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *x* and *y* are **equal**, or if they have components that are of the same *type* as each other and if those components are **equalp**; specifically, **equalp** returns *true* in the following cases:

Characters

If two *characters* are **char-equal**.

Numbers

If two *numbers* are the *same* under =.

Conses

If the two *cars* in the *conses* are **equalp** and the two *cdrs* in the *conses* are **equalp**.

Arrays

If two *arrays* have the same number of dimensions, the dimensions match, and the corresponding *active elements* are **equalp**. The *types* for which the *arrays* are *specialized* need not match; for example, a *string* and a general *array* that happens to contain the same *characters* are **equalp**. Because **equalp** performs *element*-by-*element* comparisons of *strings* and ignores the *case* of *characters*, *case* distinctions are ignored when **equalp** compares *strings*.

Structures

If two *structures* S_1 and S_2 have the same *class* and the value of each *slot* in S_1 is the *same* under **equalp** as the value of the corresponding *slot* in S_2.

Hash Tables

equalp descends *hash-tables* by first comparing the count of entries and the `:test` function; if those are the same, it compares the keys of the tables using the `:test` function and then the values of the matching keys using **equalp** recursively.

equalp does not descend any *objects* other than the ones explicitly specified above. Figure 5–13 summarizes the information given in the previous list. In addition, the figure specifies the priority of the behavior of **equalp**, with upper entries taking priority over lower ones.

Type	**Behavior**
number	uses =
character	uses **char-equal**
cons	descends
bit vector	descends
string	descends
pathname	same as **equal**
structure	descends, as described above
Other *array*	descends
hash table	descends, as described above
Other *object*	uses **eq**

Figure 5–13. Summary and priorities of behavior of equalp

Examples:

```
(equalp 'a 'b) → false
(equalp 'a 'a) → true
(equalp 3 3) → true
(equalp 3 3.0) → true
(equalp 3.0 3.0) → true
(equalp #c(3 -4) #c(3 -4)) → true
(equalp #c(3 -4.0) #c(3 -4)) → true
(equalp (cons 'a 'b) (cons 'a 'c)) → false
(equalp (cons 'a 'b) (cons 'a 'b)) → true
(equalp #\A #\A) → true
(equalp #\A #\a) → true
(equalp "Foo" "Foo") → true
(equalp "Foo" (copy-seq "Foo")) → true
```

```
 (equalp "FOO" "foo") → true

 (setq array1 (make-array 6 :element-type 'integer
                            :initial-contents '(1 1 1 3 5 7)))
→ #(1 1 1 3 5 7)
 (setq array2 (make-array 8 :element-type 'integer
                            :initial-contents '(1 1 1 3 5 7 2 6)
                            :fill-pointer 6))
→ #(1 1 1 3 5 7)
 (equalp array1 array2) → true
 (setq vector1 (vector 1 1 1 3 5 7)) → #(1 1 1 3 5 7)
 (equalp array1 vector1) → true
```

See Also:

eq, **eql**, **equal**, **=**, **string=**, **string-equal**, **char=**, **char-equal**

Notes:

Object equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound very generic, **equal** and **equalp** are not appropriate for every application.

identity *Function*

Syntax:

identity *object* → *object*

Arguments and Values:

object—an *object*.

Description:

Returns its argument *object*.

Examples:

```
 (identity 101) → 101
 (mapcan #'identity (list (list 1 2 3) '(4 5 6))) → (1 2 3 4 5 6)
```

Notes:

identity is intended for use with functions that require a *function* as an argument.

`(eql x (identity x))` returns *true* for all possible values of *x*, but `(eq x (identity x))` might return *false* when *x* is a *number* or *character*.

identity could be defined by

```
(defun identity (x) x)
```

complement *Function*

Syntax:

complement *function* → *complement-function*

Arguments and Values:

function—a *function*.

complement-function—a *function*.

Description:

Returns a *function* that takes the same *arguments* as *function*, and has the same side-effect behavior as *function*, but returns only a single value: a *generalized boolean* with the opposite truth value of that which would be returned as the *primary value* of *function*. That is, when the *function* would have returned *true* as its *primary value* the *complement-function* returns *false*, and when the *function* would have returned *false* as its *primary value* the *complement-function* returns *true*.

Examples:

```
(funcall (complement #'zerop) 1) → true
(funcall (complement #'characterp) #\A) → false
(funcall (complement #'member) 'a '(a b c)) → false
(funcall (complement #'member) 'd '(a b c)) → true
```

See Also:

not

Notes:

```
(complement x) ≡ #'(lambda (&rest arguments) (not (apply x arguments)))
```

In Common Lisp, functions with names like "*xxx*-`if-not`" are related to functions with names like "*xxx*-`if`" in that

```
(xxx-if-not f . arguments) ≡ (xxx-if (complement f) . arguments)
```

For example,

```
(find-if-not #'zerop '(0 0 3)) ≡
```

```
(find-if (complement #'zerop) '(0 0 3)) → 3
```

Note that since the "*xxx*-`if-not`" *functions* and the `:test-not` arguments have been deprecated, uses of "*xxx*-`if`" *functions* or `:test` arguments with **complement** are preferred.

constantly *Function*

Syntax:

constantly *value* → *function*

Arguments and Values:

value—an *object*.

function—a *function*.

Description:

constantly returns a *function* that accepts any number of arguments, that has no side-effects, and that always returns *value*.

Examples:

```
 (mapcar (constantly 3) '(a b c d)) → (3 3 3 3)
 (defmacro with-vars (vars &body forms)
   `((lambda ,vars ,@forms) ,@(mapcar (constantly nil) vars)))
→ WITH-VARS
 (macroexpand '(with-vars (a b) (setq a 3 b (* a a)) (list a b)))
→ ((LAMBDA (A B) (SETQ A 3 B (* A A)) (LIST A B)) NIL NIL), true
```

See Also:

identity

Notes:

constantly could be defined by:

```
 (defun constantly (object)
   #'(lambda (&rest arguments) object))
```

every, some, notevery, notany *Function*

Syntax:

every *predicate* &rest *sequences*$^+$ → *generalized-boolean*

some *predicate* &rest *sequences*$^+$ → *result*

notevery *predicate* &rest *sequences*$^+$ → *generalized-boolean*

notany *predicate* &rest *sequences*$^+$ → *generalized-boolean*

Arguments and Values:

predicate—a *designator* for a *function* of as many *arguments* as there are *sequences*.

sequence—a *sequence*.

result—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

every, **some**, **notevery**, and **notany** test *elements* of *sequences* for satisfaction of a given *predicate*. The first argument to *predicate* is an *element* of the first *sequence*; each succeeding argument is an *element* of a succeeding *sequence*.

Predicate is first applied to the elements with index 0 in each of the *sequences*, and possibly then to the elements with index 1, and so on, until a termination criterion is met or the end of the shortest of the *sequences* is reached.

every returns *false* as soon as any invocation of *predicate* returns *false*. If the end of a *sequence* is reached, **every** returns *true*. Thus, **every** returns *true* if and only if every invocation of *predicate* returns *true*.

some returns the first *non-nil* value which is returned by an invocation of *predicate*. If the end of a *sequence* is reached without any invocation of the *predicate* returning *true*, **some** returns *false*. Thus, **some** returns *true* if and only if some invocation of *predicate* returns *true*.

notany returns *false* as soon as any invocation of *predicate* returns *true*. If the end of a *sequence* is reached, **notany** returns *true*. Thus, **notany** returns *true* if and only if it is not the case that any invocation of *predicate* returns *true*.

notevery returns *true* as soon as any invocation of *predicate* returns *false*. If the end of a *sequence* is reached, **notevery** returns *false*. Thus, **notevery** returns *true* if and only if it is not the case that every invocation of *predicate* returns *true*.

Examples:

```
(every #'characterp "abc") → true
(some #'= '(1 2 3 4 5) '(5 4 3 2 1)) → true
(notevery #'< '(1 2 3 4) '(5 6 7 8) '(9 10 11 12)) → false
(notany #'> '(1 2 3 4) '(5 6 7 8) '(9 10 11 12)) → true
```

Exceptional Situations:

Should signal **type-error** if its first argument is neither a *symbol* nor a *function* or if any subsequent argument is not a *proper sequence*.

Other exceptional situations are possible, depending on the nature of the *predicate*.

See Also:

and, **or**, Section 3.6 (Traversal Rules and Side Effects)

Notes:

```
(notany predicate {sequence}*) ≡ (not (some predicate {sequence}*))
(notevery predicate {sequence}*) ≡ (not (every predicate {sequence}*))
```

and *Macro*

Syntax:

and {*form*}* → {*result*}*

Arguments and Values:

form—a *form*.

results—the *values* resulting from the evaluation of the last *form*, or the symbols **nil** or **t**.

Description:

The macro **and** evaluates each *form* one at a time from left to right. As soon as any *form* evaluates to **nil**, **and** returns **nil** without evaluating the remaining *forms*. If all *forms* but the last evaluate to *true* values, **and** returns the results produced by evaluating the last *form*.

If no *forms* are supplied, `(and)` returns **t**.

and passes back multiple values from the last *subform* but not from subforms other than the last.

Examples:

```
(if (and (>= n 0)
          (< n (length a-simple-vector))
          (eq (elt a-simple-vector n) 'foo))
     (princ "Foo!"))
```

The above expression prints `Foo!` if element `n` of `a-simple-vector` is the symbol `foo`, provided also that `n` is indeed a valid index for `a-simple-vector`. Because **and** guarantees left-to-right testing of its parts, **elt** is not called if `n` is out of range.

```
(setq temp1 1 temp2 1 temp3 1) → 1
(and (incf temp1) (incf temp2) (incf temp3)) → 2
(and (eql 2 temp1) (eql 2 temp2) (eql 2 temp3)) → true
(decf temp3) → 1
(and (decf temp1) (decf temp2) (eq temp3 'nil) (decf temp3)) → NIL
(and (eql temp1 temp2) (eql temp2 temp3)) → true
(and) → T
```

See Also:

cond, **every**, **if**, **or**, **when**

Notes:

```
(and form) ≡ (let () form)
(and form1 form2 ...) ≡ (when form1 (and form2 ...))
```

cond *Macro*

Syntax:

cond {↓*clause*}* → {*result*}*

clause::=(*test-form* {*form*}*)

Arguments and Values:

test-form—a *form*.

forms—an *implicit progn*.

results—the *values* of the *forms* in the first *clause* whose *test-form yields true*, or the *primary value* of the *test-form* if there are no *forms* in that *clause*, or else **nil** if no *test-form yields true*.

Description:

cond allows the execution of *forms* to be dependent on *test-form*.

Test-forms are evaluated one at a time in the order in which they are given in the argument list until a *test-form* is found that evaluates to *true*.

If there are no *forms* in that clause, the *primary value* of the *test-form* is returned by the **cond** *form*. Otherwise, the *forms* associated with this *test-form* are evaluated in order, left to right, as an *implicit progn*, and the *values* returned by the last *form* are returned by the **cond** *form*.

Once one *test-form* has *yielded true*, no additional *test-forms* are *evaluated*. If no *test-form yields true*, **nil** is returned.

Examples:

```
(defun select-options ()
  (cond ((= a 1) (setq a 2))
        ((= a 2) (setq a 3))
        ((and (= a 3) (floor a 2)))
        (t (floor a 3)))) → SELECT-OPTIONS
(setq a 1) → 1
(select-options) → 2
a → 2
(select-options) → 3
a → 3
(select-options) → 1
(setq a 5) → 5
(select-options) → 1, 2
```

See Also:

if, **case**.

if *Special Operator*

Syntax:

if *test-form then-form* [*else-form*] → {*result*}*

Arguments and Values:

Test-form—a *form*.

Then-form—a *form*.

Else-form—a *form*. The default is **nil**.

results—if the *test-form yielded true*, the *values* returned by the *then-form*; otherwise, the *values* returned by the *else-form*.

Description:

if allows the execution of a *form* to be dependent on a single *test-form*.

First *test-form* is evaluated. If the result is *true*, then *then-form* is selected; otherwise *else-form* is selected. Whichever form is selected is then evaluated.

Examples:

```
 (if t 1) → 1
 (if nil 1 2) → 2
 (defun test ()
   (dolist (truth-value '(t nil 1 (a b c)))
     (if truth-value (print 'true) (print 'false))
     (prin1 truth-value))) → TEST
 (test)
▷ TRUE T
▷ FALSE NIL
▷ TRUE 1
▷ TRUE (A B C)
→ NIL
```

See Also:

cond, **unless**, **when**

Notes:

```
 (if test-form then-form else-form)
 ≡ (cond (test-form then-form) (t else-form))
```

or *Macro*

Syntax:

or {*form*}* → {*results*}*

Arguments and Values:

form—a *form*.

results—the *values* or *primary value* (see below) resulting from the evaluation of the last *form* executed or **nil**.

Description:

or evaluates each *form*, one at a time, from left to right. The evaluation of all *forms* terminates when a *form* evaluates to *true* (*i.e.*, something other than **nil**).

If the *evaluation* of any *form* other than the last returns a *primary value* that is *true*, **or** immediately returns that *value* (but no additional *values*) without evaluating the remaining *forms*. If every *form* but the last returns *false* as its *primary value*, **or** returns all *values* returned by the last *form*. If no *forms* are supplied, **or** returns **nil**.

Examples:

```
(or) → NIL
(setq temp0 nil temp1 10 temp2 20 temp3 30) → 30
(or temp0 temp1 (setq temp2 37)) → 10
temp2 → 20
(or (incf temp1) (incf temp2) (incf temp3)) → 11
temp1 → 11
temp2 → 20
temp3 → 30
(or (values) temp1) → 11
(or (values temp1 temp2) temp3) → 11
(or temp0 (values temp1 temp2)) → 11, 20
(or (values temp0 temp1) (values temp2 temp3)) → 20, 30
```

See Also:

and, **some**, **unless**

when, unless *Macro*

Syntax:

when *test-form* {*form*}* → {*result*}*

unless *test-form* {*form*}* → {*result*}*

Arguments and Values:

test-form—a *form*.

forms—an *implicit progn*.

results—the *values* of the *forms* in a **when** *form* if the *test-form* *yields* *true* or in an **unless** *form* if the *test-form* *yields* *false*; otherwise **nil**.

Description:

when and **unless** allow the execution of *forms* to be dependent on a single *test-form*.

In a **when** *form*, if the *test-form* *yields* *true*, the *forms* are *evaluated* in order from left to right and the *values* returned by the *forms* are returned from the **when** *form*. Otherwise, if the *test-form* *yields* *false*, the *forms* are not *evaluated*, and the **when** *form* returns **nil**.

In an **unless** *form*, if the *test-form yields false*, the *forms* are *evaluated* in order from left to right and the *values* returned by the *forms* are returned from the **unless** *form*. Otherwise, if the *test-form yields false*, the *forms* are not *evaluated*, and the **unless** *form* returns **nil**.

Examples:

```
 (when t 'hello) → HELLO
 (unless t 'hello) → NIL
 (when nil 'hello) → NIL
 (unless nil 'hello) → HELLO
 (when t) → NIL
 (unless nil) → NIL
 (when t (prin1 1) (prin1 2) (prin1 3))
▷ 123
→ 3
 (unless t (prin1 1) (prin1 2) (prin1 3)) → NIL
 (when nil (prin1 1) (prin1 2) (prin1 3)) → NIL
 (unless nil (prin1 1) (prin1 2) (prin1 3))
▷ 123
→ 3
 (let ((x 3))
   (list (when (oddp x) (incf x) (list x))
         (when (oddp x) (incf x) (list x))
         (unless (oddp x) (incf x) (list x))
         (unless (oddp x) (incf x) (list x))
         (if (oddp x) (incf x) (list x))
         (if (oddp x) (incf x) (list x))
         (if (not (oddp x)) (incf x) (list x))
         (if (not (oddp x)) (incf x) (list x))))
→ ((4) NIL (5) NIL 6 (6) 7 (7))
```

See Also:

and, **cond**, **if**, **or**

Notes:

```
 (when test {form}+) ≡ (and test (progn {form}+))
 (when test {form}+) ≡ (cond (test {form}+))
 (when test {form}+) ≡ (if test (progn {form}+) nil)
 (when test {form}+) ≡ (unless (not test) {form}+)
 (unless test {form}+) ≡ (cond ((not test) {form}+))
 (unless test {form}+) ≡ (if test nil (progn {form}+))
 (unless test {form}+) ≡ (when (not test) {form}+)
```

case, ccase, ecase *Macro*

Syntax:

case *keyform* {↓*normal-clause*}* [↓*otherwise-clause*] → {*result*}*

ccase *keyplace* {↓*normal-clause*}* → {*result*}*

ecase *keyform* {↓*normal-clause*}* → {*result*}*

normal-clause::=(*keys* {*form*}*)

otherwise-clause::=({*otherwise* | *t*} {*form*}*)

clause::=*normal-clause* | *otherwise-clause*

Arguments and Values:

keyform—a *form*; evaluated to produce a *test-key*.

keyplace—a *form*; evaluated initially to produce a *test-key*. Possibly also used later as a *place* if no *keys* match.

test-key—an object produced by evaluating *keyform* or *keyplace*.

keys—a *designator* for a *list* of *objects*. In the case of **case**, the *symbols* **t** and **otherwise** may not be used as the *keys designator*. To refer to these *symbols* by themselves as *keys*, the designators `(t)` and `(otherwise)`, respectively, must be used instead.

forms—an *implicit progn*.

results—the *values* returned by the *forms* in the matching *clause*.

Description:

These *macros* allow the conditional execution of a body of *forms* in a *clause* that is selected by matching the *test-key* on the basis of its identity.

The *keyform* or *keyplace* is *evaluated* to produce the *test-key*.

Each of the *normal-clauses* is then considered in turn. If the *test-key* is the *same* as any *key* for that *clause*, the *forms* in that *clause* are *evaluated* as an *implicit progn*, and the *values* it returns are returned as the value of the **case**, **ccase**, or **ecase** *form*.

These *macros* differ only in their *behavior* when no *normal-clause* matches; specifically:

case

If no *normal-clause* matches, and there is an *otherwise-clause*, then that *otherwise-clause* automatically matches; the *forms* in that *clause* are *evaluated* as an *implicit progn*, and the *values* it returns are returned as the value of the **case**.

If there is no *otherwise-clause*, **case** returns **nil**.

ccase

If no *normal-clause* matches, a *correctable error* of *type* **type-error** is signaled. The offending datum is the *test-key* and the expected type is *type equivalent* to `(member` *key1 key2* `...)`. The **store-value** *restart* can be used to correct the error.

If the **store-value** *restart* is invoked, its *argument* becomes the new *test-key*, and is stored in *keyplace* as if by `(setf` *keyplace test-key*`)`. Then **ccase** starts over, considering each *clause* anew.

The subforms of *keyplace* might be evaluated again if none of the cases holds.

ecase

If no *normal-clause* matches, a *non-correctable error* of *type* **type-error** is signaled. The offending datum is the *test-key* and the expected type is *type equivalent* to `(member` *key1 key2* `...)`.

Note that in contrast with **ccase**, the caller of **ecase** may rely on the fact that **ecase** does not return if a *normal-clause* does not match.

Examples:

```
 (dolist (k '(1 2 3 :four #\v () t 'other))
    (format t "~S "
       (case k ((1 2) 'clause1)
               (3 'clause2)
               (nil 'no-keys-so-never-seen)
               ((nil) 'nilslot)
               ((:four #\v) 'clause4)
               ((t) 'tslot)
               (otherwise 'others))))
▷ CLAUSE1 CLAUSE1 CLAUSE2 CLAUSE4 CLAUSE4 NILSLOT TSLOT OTHERS
→ NIL
 (defun add-em (x) (apply #'+ (mapcar #'decode x)))
→ ADD-EM
 (defun decode (x)
   (ccase x
     ((i uno) 1)
     ((ii dos) 2)
     ((iii tres) 3)
     ((iv cuatro) 4)))
→ DECODE
```

```
 (add-em '(uno iii)) → 4
 (add-em '(uno iiii))
▷ Error: The value of X, IIII, is not I, UNO, II, DOS, III,
▷        TRES, IV, or CUATRO.
▷  1: Supply a value to use instead.
▷  2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Value to evaluate and use for X: 'IV
→ 5
```

Side Effects:

The debugger might be entered. If the **store-value** *restart* is invoked, the *value* of *keyplace* might be changed.

Affected By:

ccase and **ecase**, since they might signal an error, are potentially affected by existing *handlers* and ***debug-io***.

Exceptional Situations:

ccase and **ecase** signal an error of *type* **type-error** if no *normal-clause* matches.

See Also:

cond, **typecase**, **setf**, Section 5.1 (Generalized Reference)

Notes:

```
(case test-key
  {(({key}*) {form}*)}*)
≡
(let ((#1=#:g0001 test-key))
  (cond {((member #1# '({key}*)) {form}*)}*))
```

The specific error message used by **ecase** and **ccase** can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use **case** with an *otherwise-clause* that explicitly signals an error with an appropriate message.

typecase, ctypecase, etypecase *Macro*

Syntax:

typecase *keyform* {↓*normal-clause*}* [↓*otherwise-clause*] → {*result*}*

ctypecase *keyplace* {↓*normal-clause*}* → {*result*}*

etypecase *keyform* {↓*normal-clause*}* → {*result*}* *normal-clause*::=(*type* {*form*}*)

otherwise-clause::=({*otherwise* | *t*} {*form*}*)

clause::=*normal-clause* | *otherwise-clause*

Arguments and Values:

keyform—a *form*; evaluated to produce a *test-key*.

keyplace—a *form*; evaluated initially to produce a *test-key*. Possibly also used later as a *place* if no *types* match.

test-key—an object produced by evaluating *keyform* or *keyplace*.

type—a *type specifier*.

forms—an *implicit progn*.

results—the *values* returned by the *forms* in the matching *clause*.

Description:

These *macros* allow the conditional execution of a body of *forms* in a *clause* that is selected by matching the *test-key* on the basis of its *type*.

The *keyform* or *keyplace* is *evaluated* to produce the *test-key*.

Each of the *normal-clauses* is then considered in turn. If the *test-key* is of the *type* given by the *clauses*'s *type*, the *forms* in that *clause* are *evaluated* as an *implicit progn*, and the *values* it returns are returned as the value of the **typecase**, **ctypecase**, or **etypecase** *form*.

These *macros* differ only in their *behavior* when no *normal-clause* matches; specifically:

typecase

If no *normal-clause* matches, and there is an *otherwise-clause*, then that *otherwise-clause* automatically matches; the *forms* in that *clause* are *evaluated* as an *implicit progn*, and the *values* it returns are returned as the value of the **typecase**.

If there is no *otherwise-clause*, **typecase** returns **nil**.

ctypecase

If no *normal-clause* matches, a *correctable error* of *type* **type-error** is signaled. The offending datum is the *test-key* and the expected type is *type equivalent* to `(or` *type1 type2* `...)`. The **store-value** *restart* can be used to correct the error.

If the **store-value** *restart* is invoked, its *argument* becomes the new *test-key*, and is stored in *keyplace* as if by `(setf` *keyplace test-key*`)`. Then **ctypecase** starts over, considering each *clause* anew.

typecase, ctypecase, etypecase

If the **store-value** *restart* is invoked interactively, the user is prompted for a new *test-key* to use.

The subforms of *keyplace* might be evaluated again if none of the cases holds.

etypecase

If no *normal-clause* matches, a *non-correctable error* of *type* **type-error** is signaled. The offending datum is the *test-key* and the expected type is *type equivalent* to `(or` *type1* *type2* `...)`.

Note that in contrast with **ctypecase**, the caller of **etypecase** may rely on the fact that **etypecase** does not return if a *normal-clause* does not match.

In all three cases, is permissible for more than one *clause* to specify a matching *type*, particularly if one is a *subtype* of another; the earliest applicable *clause* is chosen.

Examples:

```
;;; (Note that the parts of this example which use TYPE-OF
;;;  are implementation-dependent.)
 (defun what-is-it (x)
   (format t "~&~S is ~A.~%"
           x (typecase x
               (float "a float")
               (null "a symbol, boolean false, or the empty list")
               (list "a list")
               (t (format nil "a(n) ~(~A~)" (type-of x))))))
→ WHAT-IS-IT
 (map 'nil #'what-is-it '(nil (a b) 7.0 7 box))
▷ NIL is a symbol, boolean false, or the empty list.
▷ (A B) is a list.
▷ 7.0 is a float.
▷ 7 is a(n) integer.
▷ BOX is a(n) symbol.
→ NIL
 (setq x 1/3)
→ 1/3
 (ctypecase x
     (integer (* x 4))
     (symbol  (symbol-value x)))
▷ Error: The value of X, 1/3, is neither an integer nor a symbol.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use instead.
▷  2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Use value: 3.7
▷ Error: The value of X, 3.7, is neither an integer nor a symbol.
▷ To continue, type :CONTINUE followed by an option number:
▷  1: Specify a value to use instead.
▷  2: Return to Lisp Toplevel.
▷ Debug> :CONTINUE 1
▷ Use value: 12
→ 48
 x → 12
```

Affected By:

ctypecase and **etypecase**, since they might signal an error, are potentially affected by existing *handlers* and ***debug-io***.

Exceptional Situations:

ctypecase and **etypecase** signal an error of *type* **type-error** if no *normal-clause* matches.

The *compiler* may choose to issue a warning of *type* **style-warning** if a *clause* will never be selected because it is completely shadowed by earlier clauses.

See Also:

case, **cond**, **setf**, Section 5.1 (Generalized Reference)

Notes:

```
(typecase test-key
  {(type {form}*)}*)
≡
(let ((#1=#:g0001 test-key))
  (cond {((typep #1# 'type) {form}*)}*))
```

The specific error message used by **etypecase** and **ctypecase** can vary between implementations. In situations where control of the specific wording of the error message is important, it is better to use **typecase** with an *otherwise-clause* that explicitly signals an error with an appropriate message.

multiple-value-bind *Macro*

Syntax:

multiple-value-bind ({*var*}*) *values-form* {*declaration*}* {*form*}*
  → {*result*}*

Arguments and Values:

var—a *symbol* naming a variable; not evaluated.

values-form—a *form*; evaluated.

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

forms—an *implicit progn*.

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

Description:

Creates new variable *bindings* for the *vars* and executes a series of *forms* that use these *bindings*.

The variable *bindings* created are lexical unless **special** declarations are specified.

Values-form is evaluated, and each of the *vars* is bound to the respective value returned by that *form*. If there are more *vars* than values returned, extra values of **nil** are given to the remaining *vars*. If there are more values than *vars*, the excess values are discarded. The *vars* are bound to the values over the execution of the *forms*, which make up an implicit **progn**. The consequences are unspecified if a type *declaration* is specified for a *var*, but the value to which that *var* is bound is not consistent with the type *declaration*.

The *scopes* of the name binding and *declarations* do not include the *values-form*.

Examples:

```
(multiple-value-bind (f r)
    (floor 130 11)
  (list f r)) → (11 9)
```

See Also:

let, **multiple-value-call**

Notes:

```
(multiple-value-bind ({var}*) values-form {form}*)
≡ (multiple-value-call #'(lambda (&optional {var}* &rest #1=#:ignore)
                            (declare (ignore #1#))
                            {form}*)
                        values-form)
```

multiple-value-call *Special Operator*

Syntax:

multiple-value-call *function-form form** → {*result*}*

Arguments and Values:

function-form—a *form*; evaluated to produce *function*.

function—a *function designator* resulting from the evaluation of *function-form*.

form—a *form*.

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

Description:

Applies *function* to a *list* of the *objects* collected from groups of *multiple values*$_2$.

multiple-value-call first evaluates the *function-form* to obtain *function*, and then evaluates each *form*. All the values of each *form* are gathered together (not just one value from each) and given as arguments to the *function*.

Examples:

```
 (multiple-value-call #'list 1 '/ (values 2 3) '/ (values) '/ (floor 2.5))
→ (1 / 2 3 / / 2 0.5)
 (+ (floor 5 3) (floor 19 4)) ≡ (+ 1 4)
→ 5
 (multiple-value-call #'+ (floor 5 3) (floor 19 4)) ≡ (+ 1 2 4 3)
→ 10
```

See Also:

multiple-value-list, **multiple-value-bind**

multiple-value-list *Macro*

Syntax:

multiple-value-list *form* → *list*

Arguments and Values:

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

list—a *list* of the *values* returned by *form*.

Description:

multiple-value-list evaluates *form* and creates a *list* of the *multiple values*$_2$ it returns.

Examples:

```
(multiple-value-list (floor -3 4)) → (-1 1)
```

See Also:

values-list, **multiple-value-call**

Notes:

multiple-value-list and **values-list** are inverses of each other.

```
(multiple-value-list form) ≡ (multiple-value-call #'list form)
```

multiple-value-prog1 *Special Operator*

Syntax:

multiple-value-prog1 *first-form* {*form*}* → *first-form-results*

Arguments and Values:

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

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

first-form-results—the *values* resulting from the *evaluation* of *first-form*.

Description:

multiple-value-prog1 evaluates *first-form* and saves all the values produced by that *form*. It then evaluates each *form* from left to right, discarding their values.

Examples:

```
(setq temp '(1 2 3)) → (1 2 3)
(multiple-value-prog1
   (values-list temp)
   (setq temp nil)
   (values-list temp)) → 1, 2, 3
```

See Also:

prog1

multiple-value-setq *Macro*

Syntax:

multiple-value-setq *vars form* → *result*

Arguments and Values:

vars—a *list* of *symbols* that are either *variable names* or *names* of *symbol macros*.

form—a *form*.

result—The *primary value* returned by the *form*.

Description:

multiple-value-setq assigns values to *vars*.

The *form* is evaluated, and each *var* is *assigned* to the corresponding *value* returned by that *form*. If there are more *vars* than *values* returned, **nil** is *assigned* to the extra *vars*. If there are more *values* than *vars*, the extra *values* are discarded.

If any *var* is the *name* of a *symbol macro*, then it is *assigned* as if by **setf**. Specifically,

(multiple-value-setq ($symbol_1$ ... $symbol_n$) *value-producing-form*)

is defined to always behave in the same way as

(values (setf (values $symbol_1$ ... $symbol_n$) *value-producing-form*))

in order that the rules for order of evaluation and side-effects be consistent with those used by **setf**. See Section 5.1.2.3 (VALUES Forms as Places).

Examples:

```
(multiple-value-setq (quotient remainder) (truncate 3.2 2)) → 1
quotient → 1
remainder → 1.2
(multiple-value-setq (a b c) (values 1 2)) → 1
a → 1
b → 2
c → NIL
(multiple-value-setq (a b) (values 4 5 6)) → 4
a → 4
b → 5
```

See Also:

setq, **symbol-macrolet**

values *Accessor*

Syntax:

values `&rest` *object* → {*object*}*

(**setf** (**values** `&rest` *place*) *new-values*)

Arguments and Values:

object—an *object*.

place—a *place*.

new-value—an *object*.

Description:

values returns the *objects* as *multiple values*$_2$.

setf of **values** is used to store the *multiple values*$_2$ *new-values* into the *places*. See Section 5.1.2.3 (VALUES Forms as Places).

Examples:

```
(values) → ⟨no values⟩
(values 1) → 1
(values 1 2) → 1, 2
(values 1 2 3) → 1, 2, 3
(values (values 1 2 3) 4 5) → 1, 4, 5
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x))) → POLAR
(multiple-value-bind (r theta) (polar 3.0 4.0)
  (vector r theta))
→ #(5.0 0.927295)
```

Sometimes it is desirable to indicate explicitly that a function returns exactly one value. For example, the function

```
(defun foo (x y)
  (floor (+ x y) y)) → FOO
```

returns two values because **floor** returns two values. It may be that the second value makes no sense, or that for efficiency reasons it is desired not to compute the second value. **values** is the standard idiom for indicating that only one value is to be returned:

```
(defun foo (x y)
  (values (floor (+ x y) y))) → FOO
```

This works because **values** returns exactly one value for each of *args*; as for any function call, if any of *args* produces more than one value, all but the first are discarded.

See Also:

values-list, **multiple-value-bind**, **multiple-values-limit**, Section 3.1 (Evaluation)

Notes:

Since **values** is a *function*, not a *macro* or *special form*, it receives as *arguments* only the *primary values* of its *argument forms*.

values-list *Function*

Syntax:

values-list *list* → {*element*}*

Arguments and Values:

list—a *list*.

elements—the *elements* of the *list*.

Description:

Returns the *elements* of the *list* as *multiple values*$_2$.

Examples:

```
(values-list nil) → ⟨no values⟩
(values-list '(1)) → 1
(values-list '(1 2)) → 1, 2
(values-list '(1 2 3)) → 1, 2, 3
```

Exceptional Situations:

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

See Also:

multiple-value-bind, **multiple-value-list**, **multiple-values-limit**, **values**

Notes:

```
(values-list list) ≡ (apply #'values list)
```

`(equal x (multiple-value-list (values-list x)))` returns *true* for all *lists* *x*.

multiple-values-limit *Constant Variable*

Constant Value:

An *integer* not smaller than 20, the exact magnitude of which is *implementation-dependent*.

Description:

The upper exclusive bound on the number of *values* that may be returned from a *function*, bound or assigned by **multiple-value-bind** or **multiple-value-setq**, or passed as a first argument to **nth-value**. (If these individual limits might differ, the minimum value is used.)

See Also:

lambda-parameters-limit, **call-arguments-limit**

Notes:

Implementors are encouraged to make this limit as large as possible.

nth-value *Macro*

Syntax:

nth-value *n form* → *object*

Arguments and Values:

n—a non-negative *integer*; evaluated.

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

object—an *object*.

Description:

Evaluates *n* and then *form*, returning as its only value the *n*th value *yielded* by *form*, or **nil** if *n* is greater than or equal to the number of *values* returned by *form*. (The first returned value is numbered 0.)

Examples:

```
(nth-value 0 (values 'a 'b)) → A
(nth-value 1 (values 'a 'b)) → B
(nth-value 2 (values 'a 'b)) → NIL
(let* ((x 83927472397238947423879243432432432)
       (y 32423489732)
```

```
        (a (nth-value 1 (floor x y)))
        (b (mod x y)))
    (values a b (= a b)))
→ 3332987528, 3332987528, true
```

See Also:

multiple-value-list, **nth**

Notes:

Operationally, the following relationship is true, although **nth-value** might be more efficient in some *implementations* because, for example, some *consing* might be avoided.

(nth-value *n* *form*) ≡ (nth *n* (multiple-value-list *form*))

prog, prog∗ *Macro*

Syntax:

prog ({*var* | (*var* [*init-form*])}*) {*declaration*}* {*tag* | *statement*}*
→ {*result*}*

prog* ({*var* | (*var* [*init-form*])}*) {*declaration*}* {*tag* | *statement*}*
→ {*result*}*

Arguments and Values:

var—variable name.

init-form—a *form*.

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

tag—a *go tag*; not evaluated.

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

results—**nil** if a *normal return* occurs, or else, if an *explicit return* occurs, the *values* that were transferred.

Description:

Three distinct operations are performed by **prog** and **prog***: they bind local variables, they permit use of the **return** statement, and they permit use of the **go** statement. A typical **prog** looks like this:

```
(prog (var1 var2 (var3 init-form-3) var4 (var5 init-form-5))
      {declaration}*
```

prog, prog∗

```
         statement1
     tag1
         statement2
         statement3
         statement4
     tag2
         statement5
         ...
         )
```

For **prog**, *init-forms* are evaluated first, in the order in which they are supplied. The *vars* are then bound to the corresponding values in parallel. If no *init-form* is supplied for a given *var*, that *var* is bound to **nil**.

The body of **prog** is executed as if it were a **tagbody** *form*; the **go** statement can be used to transfer control to a *tag*. *Tags* label *statements*.

prog implicitly establishes a **block** named **nil** around the entire **prog** *form*, so that **return** can be used at any time to exit from the **prog** *form*.

The difference between **prog*** and **prog** is that in **prog*** the *binding* and initialization of the *vars* is done *sequentially*, so that the *init-form* for each one can use the values of previous ones.

Examples:

```
(prog* ((y z) (x (car y)))
       (return x))
```

returns the *car* of the value of z.

```
 (setq a 1) → 1
 (prog ((a 2) (b a)) (return (if (= a b) '= '/=))) → /=
 (prog* ((a 2) (b a)) (return (if (= a b) '= '/=))) → =
 (prog () 'no-return-value) → NIL

 (defun king-of-confusion (w)
   "Take a cons of two lists and make a list of conses.
    Think of this function as being like a zipper."
   (prog (x y z)          ;Initialize x, y, z to NIL
        (setq y (car w) z (cdr w))
    loop
        (cond ((null y) (return x))
              ((null z) (go err)))
    rejoin
        (setq x (cons (cons (car y) (car z)) x))
        (setq y (cdr y) z (cdr z))
        (go loop)
    err
```

```
        (cerror "Will self-pair extraneous items"
                "Mismatch - gleep!  ~S" y)
        (setq z y)
        (go rejoin))) → KING-OF-CONFUSION
```

This can be accomplished more perspicuously as follows:

```
 (defun prince-of-clarity (w)
   "Take a cons of two lists and make a list of conses.
    Think of this function as being like a zipper."
   (do ((y (car w) (cdr y))
        (z (cdr w) (cdr z))
        (x '() (cons (cons (car y) (car z)) x)))
       ((null y) x)
     (when (null z)
       (cerror "Will self-pair extraneous items"
              "Mismatch - gleep!  ~S" y)
       (setq z y)))) → PRINCE-OF-CLARITY
```

See Also:

block, **let**, **tagbody**, **go**, **return**, Section 3.1 (Evaluation)

Notes:

prog can be explained in terms of **block**, **let**, and **tagbody** as follows:

(prog *variable-list declaration* . *body*)
   ≡ (block nil (let *variable-list declaration* (tagbody . *body*)))

prog1, prog2 *Macro*

Syntax:

prog1 *first-form* {*form*}* → *result-1*

prog2 *first-form second-form* {*form*}* → *result-2*

Arguments and Values:

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

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

forms—an *implicit progn*; evaluated as described below.

result-1—the *primary value* resulting from the *evaluation* of *first-form*.

prog1, prog2

result-2—the *primary value* resulting from the *evaluation* of *second-form*.

Description:

prog1 *evaluates* *first-form* and then *forms*, *yielding* as its only *value* the *primary value yielded* by *first-form*.

prog2 *evaluates* *first-form*, then *second-form*, and then *forms*, *yielding* as its only *value* the *primary value yielded* by *first-form*.

Examples:

```
 (setq temp 1) → 1
 (prog1 temp (print temp) (incf temp) (print temp))
▷ 1
▷ 2
→ 1
 (prog1 temp (setq temp nil)) → 2
 temp → NIL
 (prog1 (values 1 2 3) 4) → 1
 (setq temp (list 'a 'b 'c))
 (prog1 (car temp) (setf (car temp) 'alpha)) → A
 temp → (ALPHA B C)
 (flet ((swap-symbol-values (x y)
          (setf (symbol-value x)
                (prog1 (symbol-value y)
                       (setf (symbol-value y) (symbol-value x))))))
   (let ((*foo* 1) (*bar* 2))
     (declare (special *foo* *bar*))
     (swap-symbol-values '*foo* '*bar*)
     (values *foo* *bar*)))
→ 2, 1
 (setq temp 1) → 1
 (prog2 (incf temp) (incf temp) (incf temp)) → 3
 temp → 4
 (prog2 1 (values 2 3 4) 5) → 2
```

See Also:

multiple-value-prog1, **progn**

Notes:

prog1 and **prog2** are typically used to *evaluate* one or more *forms* with side effects and return a *value* that must be computed before some or all of the side effects happen.

```
(prog1 {form}*) ≡ (values (multiple-value-prog1 {form}*))
(prog2 form1 {form}*) ≡ (let () form1 (prog1 {form}*))
```

progn *Special Operator*

Syntax:

progn {*form*}* → {*result*}*

Arguments and Values:

forms—an *implicit progn*.

results—the *values* of the *forms*.

Description:

progn evaluates *forms*, in the order in which they are given.

The values of each *form* but the last are discarded.

If **progn** appears as a *top level form*, then all *forms* within that **progn** are considered by the compiler to be *top level forms*.

Examples:

```
(progn) → NIL
(progn 1 2 3) → 3
(progn (values 1 2 3)) → 1, 2, 3
(setq a 1) → 1
(if a
     (progn (setq a nil) 'here)
     (progn (setq a t) 'there)) → HERE
a → NIL
```

See Also:

prog1, **prog2**, Section 3.1 (Evaluation)

Notes:

Many places in Common Lisp involve syntax that uses *implicit progns*. That is, part of their syntax allows many *forms* to be written that are to be evaluated sequentially, discarding the results of all *forms* but the last and returning the results of the last *form*. Such places include, but are not limited to, the following: the body of a *lambda expression*; the bodies of various control and conditional *forms* (*e.g.*, **case**, **catch**, **progn**, and **when**).

define-modify-macro

define-modify-macro *Macro*

Syntax:

define-modify-macro *name lambda-list function* [*documentation*] → *name*

Arguments and Values:

name—a *symbol*.

lambda-list—a *define-modify-macro lambda list*

function—a *symbol*.

documentation—a *string*; not evaluated.

Description:

define-modify-macro defines a *macro* named *name* to *read* and *write* a *place*.

The arguments to the new *macro* are a *place*, followed by the arguments that are supplied in *lambda-list*. *Macros* defined with **define-modify-macro** correctly pass the *environment parameter* to **get-setf-expansion**.

When the *macro* is invoked, *function* is applied to the old contents of the *place* and the *lambda-list* arguments to obtain the new value, and the *place* is updated to contain the result.

Except for the issue of avoiding multiple evaluation (see below), the expansion of a **define-modify-macro** is equivalent to the following:

```
(defmacro name (reference . lambda-list)
  documentation
  `(setf ,reference
         (function ,reference ,arg1 ,arg2 ...)))
```

where *arg1*, *arg2*, ..., are the parameters appearing in *lambda-list*; appropriate provision is made for a *rest parameter*.

The *subforms* of the macro calls defined by **define-modify-macro** are evaluated as specified in Section 5.1.1.1 (Evaluation of Subforms to Places).

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

If a **define-modify-macro** *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.

Examples:

```
(define-modify-macro appendf (&rest args)
   append "Append onto list") → APPENDF
(setq x '(a b c) y x) → (A B C)
(appendf x '(d e f) '(1 2 3)) → (A B C D E F 1 2 3)
x → (A B C D E F 1 2 3)
y → (A B C)
(define-modify-macro new-incf (&optional (delta 1)) +)
(define-modify-macro unionf (other-set &rest keywords) union)
```

Side Effects:

A macro definition is assigned to *name*.

See Also:

defsetf, **define-setf-expander**, **documentation**, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

defsetf *Macro*

Syntax:

The "short form":

defsetf *access-fn update-fn* [*documentation*]
  → *access-fn*

The "long form":

defsetf *access-fn lambda-list* ({*store-variable*}*) ⟦{*declaration*}* | *documentation*⟧ {*form*}*
  → *access-fn*

Arguments and Values:

access-fn—a *symbol* which names a *function* or a *macro*.

update-fn—a *symbol* naming a *function* or *macro*.

lambda-list—a *defsetf lambda list*.

store-variable—a *symbol* (a *variable name*).

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

documentation—a *string*; not evaluated.

form—a *form*.

defsetf

Description:

defsetf defines how to **setf** a *place* of the form `(`*access-fn* `...)` for relatively simple cases. (See **define-setf-expander** for more general access to this facility.) It must be the case that the *function* or *macro* named by *access-fn* evaluates all of its arguments.

defsetf may take one of two forms, called the "short form" and the "long form," which are distinguished by the *type* of the second *argument*.

When the short form is used, *update-fn* must name a *function* (or *macro*) that takes one more argument than *access-fn* takes. When **setf** is given a *place* that is a call on *access-fn*, it expands into a call on *update-fn* that is given all the arguments to *access-fn* and also, as its last argument, the new value (which must be returned by *update-fn* as its value).

The long form **defsetf** resembles **defmacro**. The *lambda-list* describes the arguments of *access-fn*. The *store-variables* describe the value or values to be stored into the *place*. The *body* must compute the expansion of a **setf** of a call on *access-fn*. The expansion function is defined in the same *lexical environment* in which the **defsetf** *form* appears.

During the evaluation of the *forms*, the variables in the *lambda-list* and the *store-variables* are bound to names of temporary variables, generated as if by **gensym** or **gentemp**, that will be bound by the expansion of **setf** to the values of those *subforms*. This binding permits the *forms* to be written without regard for order-of-evaluation issues. **defsetf** arranges for the temporary variables to be optimized out of the final result in cases where that is possible.

The body code in **defsetf** is implicitly enclosed in a *block* whose name is *access-fn*

defsetf ensures that *subforms* of the *place* are evaluated exactly once.

Documentation is attached to *access-fn* as a *documentation string* of kind **setf**.

If a **defsetf** *form* appears as a *top level form*, the *compiler* must make the *setf expander* available so that it may be used to expand calls to **setf** later on in the *file*. Users must ensure that the *forms*, if any, can be evaluated at compile time if the *access-fn* is used in a *place* later in the same *file*. The *compiler* must make these *setf expanders* available to compile-time calls to **get-setf-expansion** when its *environment* argument is a value received as the *environment parameter* of a *macro*.

Examples:

The effect of

```
(defsetf symbol-value set)
```

is built into the Common Lisp system. This causes the form `(setf (symbol-value foo) fu)` to expand into `(set foo fu)`.

Note that

```
(defsetf car rplaca)
```

would be incorrect because **rplaca** does not return its last argument.

```
(defun middleguy (x) (nth (truncate (1- (list-length x)) 2) x)) → MIDDLEGUY
(defun set-middleguy (x v)
   (unless (null x)
     (rplaca (nthcdr (truncate (1- (list-length x)) 2) x) v))
   v) → SET-MIDDLEGUY
(defsetf middleguy set-middleguy) → MIDDLEGUY
(setq a (list 'a 'b 'c 'd)
      b (list 'x)
      c (list 1 2 3 (list 4 5 6) 7 8 9)) → (1 2 3 (4 5 6) 7 8 9)
(setf (middleguy a) 3) → 3
(setf (middleguy b) 7) → 7
(setf (middleguy (middleguy c)) 'middleguy-symbol) → MIDDLEGUY-SYMBOL
a → (A 3 C D)
b → (7)
c → (1 2 3 (4 MIDDLEGUY-SYMBOL 6) 7 8 9)
```

An example of the use of the long form of **defsetf**:

```
(defsetf subseq (sequence start &optional end) (new-sequence)
  `(progn (replace ,sequence ,new-sequence
                   :start1 ,start :end1 ,end)
          ,new-sequence)) → SUBSEQ

(defvar *xy* (make-array '(10 10)))
(defun xy (&key ((x x) 0) ((y y) 0)) (aref *xy* x y)) → XY
(defun set-xy (new-value &key ((x x) 0) ((y y) 0))
  (setf (aref *xy* x y) new-value)) → SET-XY
(defsetf xy (&key ((x x) 0) ((y y) 0)) (store)
  `(set-xy ,store 'x ,x 'y ,y)) → XY
(get-setf-expansion '(xy a b))
→ (#:t0 #:t1),
   (a b),
   (#:store),
   ((lambda (&key ((x #:x)) ((y #:y)))
      (set-xy #:store 'x #:x 'y #:y))
    #:t0 #:t1),
   (xy #:t0 #:t1)
(xy 'x 1) → NIL
(setf (xy 'x 1) 1) → 1
(xy 'x 1) → 1
(let ((a 'x) (b 'y))
  (setf (xy a 1 b 2) 3)
  (setf (xy b 5 a 9) 14))
→ 14
(xy 'y 0 'x 1) → 1
```

```
(xy 'x 1 'y 2) → 3
```

See Also:

documentation, **setf**, **define-setf-expander**, **get-setf-expansion**, Section 5.1 (Generalized Reference), Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

forms must include provision for returning the correct value (the value or values of *store-variable*). This is handled by *forms* rather than by **defsetf** because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the *place* and returns the correct value.

A **setf** of a call on *access-fn* also evaluates all of *access-fn*'s arguments; it cannot treat any of them specially. This means that **defsetf** cannot be used to describe how to store into a *generalized reference* to a byte, such as `(ldb field reference)`. **define-setf-expander** is used to handle situations that do not fit the restrictions imposed by **defsetf** and gives the user additional control.

define-setf-expander *Macro*

Syntax:

define-setf-expander *access-fn lambda-list*
⟦{*declaration*}* | *documentation*⟧ {*form*}*

→ *access-fn*

Arguments and Values:

access-fn—a *symbol* that *names* a *function* or *macro*.

lambda-list – *macro lambda list*.

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

documentation—a *string*; not evaluated.

forms—an *implicit progn*.

Description:

define-setf-expander specifies the means by which **setf** updates a *place* that is referenced by *access-fn*.

When **setf** is given a *place* that is specified in terms of *access-fn* and a new value for the *place*, it is expanded into a form that performs the appropriate update.

The *lambda-list* supports destructuring. See Section 3.4.4 (Macro Lambda Lists).

Documentation is attached to *access-fn* as a *documentation string* of kind **setf**.

Forms constitute the body of the *setf expander* definition and must compute the *setf expansion* for a call on **setf** that references the *place* by means of the given *access-fn*. The *setf expander* function is defined in the same *lexical environment* in which the **define-setf-expander** *form* appears. While *forms* are being executed, the variables in *lambda-list* are bound to parts of the *place form*. The body *forms* (but not the *lambda-list*) in a **define-setf-expander** *form* are implicitly enclosed in a *block* whose name is *access-fn*.

The evaluation of *forms* must result in the five values described in Section 5.1.1.2 (Setf Expansions).

If a **define-setf-expander** *form* appears as a *top level form*, the *compiler* must make the *setf expander* available so that it may be used to expand calls to **setf** later on in the *file*. *Programmers* must ensure that the *forms* can be evaluated at compile time if the *access-fn* is used in a *place* later in the same *file*. The *compiler* must make these *setf expanders* available to compile-time calls to **get-setf-expansion** when its *environment* argument is a value received as the *environment parameter* of a *macro*.

Examples:

```
 (defun lastguy (x) (car (last x))) → LASTGUY
 (define-setf-expander lastguy (x &environment env)
   "Set the last element in a list to the given value."
   (multiple-value-bind (dummies vals newval setter getter)
       (get-setf-expansion x env)
     (let ((store (gensym)))
       (values dummies
               vals
               `(,store)
               `(progn (rplaca (last ,getter) ,store) ,store)
               `(lastguy ,getter))))) → LASTGUY
 (setq a (list 'a 'b 'c 'd)
       b (list 'x)
       c (list 1 2 3 (list 4 5 6))) → (1 2 3 (4 5 6))
 (setf (lastguy a) 3) → 3
 (setf (lastguy b) 7) → 7
 (setf (lastguy (lastguy c)) 'lastguy-symbol) → LASTGUY-SYMBOL
 a → (A B C 3)
 b → (7)
 c → (1 2 3 (4 5 LASTGUY-SYMBOL))


;;; Setf expander for the form (LDB bytespec int).
;;; Recall that the int form must itself be suitable for SETF.
 (define-setf-expander ldb (bytespec int &environment env)
   (multiple-value-bind (temps vals stores
                          store-form access-form)
       (get-setf-expansion int env);Get setf expansion for int.
     (let ((btemp (gensym))     ;Temp var for byte specifier.
           (store (gensym))     ;Temp var for byte to store.
           (stemp (first stores))) ;Temp var for int to store.
       (if (cdr stores) (error "Can't expand this."))
;;; Return the setf expansion for LDB as five values.
       (values (cons btemp temps)       ;Temporary variables.
               (cons bytespec vals)     ;Value forms.
               (list store)             ;Store variables.
               `(let ((,stemp (dpb ,store ,btemp ,access-form)))
                  ,store-form
                  ,store)               ;Storing form.
               `(ldb ,btemp ,access-form) ;Accessing form.
              ))))
```

See Also:

setf, **defsetf**, **documentation**, **get-setf-expansion**, Section 3.4.11 (Syntactic Interaction of Documentation Strings and Declarations)

Notes:

define-setf-expander differs from the long form of **defsetf** in that while the body is being executed the *variables* in *lambda-list* are bound to parts of the *place form*, not to temporary variables that will be bound to the values of such parts. In addition, **define-setf-expander** does not have **defsetf**'s restriction that *access-fn* must be a *function* or a function-like *macro*; an arbitrary **defmacro** destructuring pattern is permitted in *lambda-list*.

get-setf-expansion *Function*

Syntax:

get-setf-expansion *place* `&optional` *environment*
→ *vars*, *vals*, *store-vars*, *writer-form*, *reader-form*

Arguments and Values:

place—a *place*.

environment—an *environment object*.

vars, *vals*, *store-vars*, *writer-form*, *reader-form*—a *setf expansion*.

Description:

Determines five values constituting the *setf expansion* for *place* in *environment*; see Section 5.1.1.2 (Setf Expansions).

If *environment* is not supplied or **nil**, the environment is the *null lexical environment*.

Examples:

```
 (get-setf-expansion 'x)
→ NIL, NIL, (#:G0001), (SETQ X #:G0001), X


;;; This macro is like POP

 (defmacro xpop (place &environment env)
   (multiple-value-bind (dummies vals new setter getter)
                        (get-setf-expansion place env)
      `(let* (,@(mapcar #'list dummies vals) (,(car new) ,getter))
         (if (cdr new) (error "Can't expand this."))
         (prog1 (car ,(car new))
                (setq ,(car new) (cdr ,(car new)))
                ,setter))))

 (defsetf frob (x) (value)
     `(setf (car ,x) ,value)) → FROB
;;; The following is an error; an error might be signaled at macro expansion time
 (flet ((frob (x) (cdr x)))  ;Invalid
   (xpop (frob z)))
```

See Also:

defsetf, **define-setf-expander**, **setf**

Notes:

Any *compound form* is a valid *place*, since any *compound form* whose *operator* *f* has no *setf expander* are expanded into a call to `(setf f)`.

setf, psetf *Macro*

Syntax:

setf {↓*pair*}* → {*result*}*

setf, psetf

psetf {↓*pair*}* → **nil**

pair::=*place newvalue*

Arguments and Values:

place—a *place*.

newvalue—a *form*.

results—the *multiple values*$_2$ returned by the storing form for the last *place*, or **nil** if there are no *pairs*.

Description:

setf changes the *value* of *place* to be *newvalue*.

`(setf place newvalue)` expands into an update form that stores the result of evaluating *newvalue* into the location referred to by *place*. Some *place* forms involve uses of accessors that take optional arguments. Whether those optional arguments are permitted by **setf**, or what their use is, is up to the **setf** expander function and is not under the control of **setf**. The documentation for any *function* that accepts **&optional**, **&rest**, or `&key` arguments and that claims to be usable with **setf** must specify how those arguments are treated.

If more than one *pair* is supplied, the *pairs* are processed sequentially; that is,

```
(setf place-1 newvalue-1
      place-2 newvalue-2
      ...
      place-N newvalue-N)
```

is precisely equivalent to

```
(progn (setf place-1 newvalue-1)
       (setf place-2 newvalue-2)
       ...
       (setf place-N newvalue-N))
```

For **psetf**, if more than one *pair* is supplied then the assignments of new values to places are done in parallel. More precisely, all *subforms* (in both the *place* and *newvalue forms*) that are to be evaluated are evaluated from left to right; after all evaluations have been performed, all of the assignments are performed in an unpredictable order.

For detailed treatment of the expansion of **setf** and **psetf**, see Section 5.1.2 (Kinds of Places).

Examples:

```
(setq x (cons 'a 'b) y (list 1 2 3)) → (1 2 3)
(setf (car x) 'x (cadr y) (car x) (cdr x) y) → (1 X 3)
x → (X 1 X 3)
```

```
 y → (1 X 3)
 (setq x (cons 'a 'b) y (list 1 2 3)) → (1 2 3)
 (psetf (car x) 'x (cadr y) (car x) (cdr x) y) → NIL
 x → (X 1 A 3)
 y → (1 A 3)
```

Affected By:

define-setf-expander, **defsetf**, ***macroexpand-hook***

See Also:

define-setf-expander, **defsetf**, **macroexpand-1**, **rotatef**, **shiftf**, Section 5.1 (Generalized Reference)

shiftf *Macro*

Syntax:

shiftf $\{$*place*$\}^+$ *newvalue* → *old-value-1*

Arguments and Values:

place—a *place*.

newvalue—a *form*; evaluated.

old-value-1—an *object* (the old *value* of the first *place*).

Description:

shiftf modifies the values of each *place* by storing *newvalue* into the last *place*, and shifting the values of the second through the last *place* into the remaining *places*.

If *newvalue* produces more values than there are store variables, the extra values are ignored. If *newvalue* produces fewer values than there are store variables, the missing values are set to **nil**.

In the form `(shiftf` *place1 place2 ... placen newvalue*`)`, the values in *place1* through *placen* are *read* and saved, and *newvalue* is evaluated, for a total of `n`+1 values in all. Values 2 through `n`+1 are then stored into *place1* through *placen*, respectively. It is as if all the *places* form a shift register; the *newvalue* is shifted in from the right, all values shift over to the left one place, and the value shifted out of *place1* is returned.

For information about the *evaluation* of *subforms* of *places*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

Examples:

```
 (setq x (list 1 2 3) y 'trash) → TRASH
```

```
(shiftf y x (cdr x) '(hi there)) → TRASH
x → (2 3)
y → (1 HI THERE)

(setq x (list 'a 'b 'c)) → (A B C)
(shiftf (cadr x) 'z) → B
x → (A Z C)
(shiftf (cadr x) (cddr x) 'q) → Z
x → (A (C) . Q)
(setq n 0) → 0
(setq x (list 'a 'b 'c 'd)) → (A B C D)
(shiftf (nth (setq n (+ n 1)) x) 'z) → B
x → (A Z C D)
```

Affected By:

define-setf-expander, **defsetf**, ***macroexpand-hook***

See Also:

setf, **rotatef**, Section 5.1 (Generalized Reference)

Notes:

The effect of `(shiftf` *place1 place2* ... *placen newvalue*`)` is roughly equivalent to

```
(let ((var1 place1)
      (var2 place2)
      ...
      (varn placen)
      (var0 newvalue))
  (setf place1 var2)
  (setf place2 var3)
  ...
  (setf placen var0)
  var1)
```

except that the latter would evaluate any *subforms* of each `place` twice, whereas **shiftf** evaluates them once. For example,

```
(setq n 0) → 0
(setq x (list 'a 'b 'c 'd)) → (A B C D)
(prog1 (nth (setq n (+ n 1)) x)
       (setf (nth (setq n (+ n 1)) x) 'z)) → B
x → (A B Z D)
```

rotatef *Macro*

Syntax:

rotatef {*place*}* → **nil**

Arguments and Values:

place—a *place*.

Description:

rotatef modifies the values of each *place* by rotating values from one *place* into another.

If a *place* produces more values than there are store variables, the extra values are ignored. If a *place* produces fewer values than there are store variables, the missing values are set to **nil**.

In the form `(rotatef` *place1 place2* ... *placen*`)`, the values in *place1* through *placen* are *read* and *written*. Values 2 through *n* and value 1 are then stored into *place1* through *placen*. It is as if all the places form an end-around shift register that is rotated one place to the left, with the value of *place1* being shifted around the end to *placen*.

For information about the *evaluation* of *subforms* of *places*, see Section 5.1.1.1 (Evaluation of Subforms to Places).

Examples:

```
(let ((n 0)
       (x (list 'a 'b 'c 'd 'e 'f 'g)))
   (rotatef (nth (incf n) x)
            (nth (incf n) x)
            (nth (incf n) x))
   x) → (A C D B E F G)
```

See Also:

define-setf-expander, **defsetf**, **setf**, **shiftf**, ***macroexpand-hook***, Section 5.1 (Generalized Reference)

Notes:

The effect of `(rotatef` *place1 place2* ... *placen*`)` is roughly equivalent to

```
(psetf place1 place2
       place2 place3
       ...
       placen place1)
```

except that the latter would evaluate any *subforms* of each `place` twice, whereas **rotatef** evaluates them once.

control-error *Condition Type*

Class Precedence List:

control-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **control-error** consists of error conditions that result from invalid dynamic transfers of control in a program. The errors that result from giving **throw** a tag that is not active or from giving **go** or **return-from** a tag that is no longer dynamically available are of *type* **control-error**.

program-error *Condition Type*

Class Precedence List:

program-error, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **program-error** consists of error conditions related to incorrect program syntax. The errors that result from naming a *go tag* or a *block tag* that is not lexically apparent are of *type* **program-error**.

undefined-function *Condition Type*

Class Precedence List:

undefined-function, **cell-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **undefined-function** consists of *error conditions* that represent attempts to *read* the definition of an *undefined function*.

The name of the cell (see **cell-error**) is the *function name* which was *funbound*.

See Also:

cell-error-name