Programming Language—Common Lisp

10. Symbols

10.1 Symbol Concepts

Figure 10–1 lists some *defined names* that are applicable to the *property lists* of *symbols*.

get	**remprop**	**symbol-plist**

Figure 10–1. Property list defined names

Figure 10–2 lists some *defined names* that are applicable to the creation of and inquiry about *symbols*.

copy-symbol	**keywordp**	**symbol-package**
gensym	**make-symbol**	**symbol-value**
gentemp	**symbol-name**	

Figure 10–2. Symbol creation and inquiry defined names

symbol *System Class*

Class Precedence List:

symbol, **t**

Description:

Symbols are used for their *object* identity to name various entities in Common Lisp, including (but not limited to) linguistic entities such as *variables* and *functions*.

Symbols can be collected together into *packages*. A *symbol* is said to be *interned* in a *package* if it is *accessible* in that *package*; the same *symbol* can be *interned* in more than one *package*. If a *symbol* is not *interned* in any *package*, it is called *uninterned*.

An *interned symbol* is uniquely identifiable by its *name* from any *package* in which it is *accessible*.

Symbols have the following attributes. For historically reasons, these are sometimes referred to as *cells*, although the actual internal representation of *symbols* and their attributes is *implementation-dependent*.

Name

The *name* of a *symbol* is a *string* used to identify the *symbol*. Every *symbol* has a *name*, and the consequences are undefined if that *name* is altered. The *name* is used as part of the external, printed representation of the *symbol*; see Section 2.1 (Character Syntax). The *function* **symbol-name** returns the *name* of a given *symbol*. A *symbol* may have any *character* in its *name*.

Package

The *object* in this *cell* is called the *home package* of the *symbol*. If the *home package* is **nil**, the *symbol* is sometimes said to have no *home package*.

When a *symbol* is first created, it has no *home package*. When it is first *interned*, the *package* in which it is initially *interned* becomes its *home package*. The *home package* of a *symbol* can be *accessed* by using the *function* **symbol-package**.

If a *symbol* is *uninterned* from the *package* which is its *home package*, its *home package* is set to **nil**. Depending on whether there is another *package* in which the *symbol* is *interned*, the symbol might or might not really be an *uninterned symbol*. A *symbol* with no *home package* is therefore called *apparently uninterned*.

The consequences are undefined if an attempt is made to alter the *home package* of a *symbol* external in the `COMMON-LISP` *package* or the `KEYWORD` *package*.

Property list

The *property list* of a *symbol* provides a mechanism for associating named attributes with that *symbol*. The operations for adding and removing entries are *destructive* to the *property list*. Common Lisp provides *operators* both for direct manipulation of *property list objects* (*e.g.*, see **getf**, **remf**, and **symbol-plist**) and for implicit manipulation of a *symbol*'s *property list* by reference to the *symbol* (*e.g.*, see **get** and **remprop**). The *property list* associated with a *fresh symbol* is initially *null*.

Value

If a symbol has a value attribute, it is said to be *bound*, and that fact can be detected by the *function* **boundp**. The *object* contained in the *value cell* of a *bound symbol* is the *value* of the *global variable* named by that *symbol*, and can be *accessed* by the *function* **symbol-value**. A *symbol* can be made to be *unbound* by the *function* **makunbound**.

The consequences are undefined if an attempt is made to change the *value* of a *symbol* that names a *constant variable*, or to make such a *symbol* be *unbound*.

Function

If a symbol has a function attribute, it is said to be *fbound*, and that fact can be detected by the *function* **fboundp**. If the *symbol* is the *name* of a *function* in the *global environment*, the *function cell* contains the *function*, and can be *accessed* by the *function* **symbol-function**. If the *symbol* is the *name* of either a *macro* in the *global environment* (see **macro-function**) or a *special operator* (see **special-operator-p**), the *symbol* is *fbound*, and can be *accessed* by the *function* **symbol-function**, but the *object* which the *function cell* contains is of *implementation-dependent type* and purpose. A *symbol* can be made to be *funbound* by the *function* **fmakunbound**.

The consequences are undefined if an attempt is made to change the *functional value* of a *symbol* that names a *special form*.

Operations on a *symbol*'s *value cell* and *function cell* are sometimes described in terms of their effect on the *symbol* itself, but the user should keep in mind that there is an intimate relationship between the contents of those *cells* and the *global variable* or global *function* definition, respectively.

Symbols are used as identifiers for *lexical variables* and lexical *function* definitions, but in that role, only their *object* identity is significant. Common Lisp provides no operation on a *symbol* that can have any effect on a *lexical variable* or on a lexical *function* definition.

See Also:

Section 2.3.4 (Symbols as Tokens), Section 2.3.1.1 (Potential Numbers as Tokens), Section 22.1.3.6 (Printing Symbols)

keyword *Type*

Supertypes:

keyword, **symbol**, **t**

Description:

The *type* **keyword** includes all *symbols interned* the `KEYWORD` *package*.

Interning a *symbol* in the `KEYWORD` *package* has three automatic effects:

1. It causes the *symbol* to become *bound* to itself.

2. It causes the *symbol* to become an *external symbol* of the `KEYWORD` *package*.

3. It causes the *symbol* to become a *constant variable*.

See Also:

keywordp

symbolp *Function*

Syntax:

symbolp *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

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

Examples:

```
(symbolp 'elephant) → true
(symbolp 12) → false
(symbolp nil) → true
(symbolp '()) → true
(symbolp :test) → true
(symbolp "hello") → false
```

See Also:

keywordp, **symbol**, **typep**

Notes:

```
(symbolp object) ≡ (typep object 'symbol)
```

keywordp *Function*

Syntax:

keywordp *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns *true* if *object* is a $keyword_1$; otherwise, returns *false*.

Examples:

```
(keywordp 'elephant) → false
(keywordp 12) → false
(keywordp :test) → true
(keywordp ':test) → true
(keywordp nil) → false
(keywordp :nil) → true
(keywordp '(:test)) → false
(keywordp "hello") → false
(keywordp ":hello") → false
(keywordp '&optional) → false
```

See Also:

constantp, **keyword**, **symbolp**, **symbol-package**

make-symbol *Function*

Syntax:

make-symbol *name* → *new-symbol*

Arguments and Values:

name—a *string*.

new-symbol—a *fresh*, *uninterned symbol*.

Description:

make-symbol creates and returns a *fresh*, *uninterned symbol* whose *name* is the given *name*. The *new-symbol* is neither *bound* nor *fbound* and has a *null property list*.

It is *implementation-dependent* whether the *string* that becomes the *new-symbol*'s *name* is the given *name* or a copy of it. Once a *string* has been given as the *name argument* to *make-symbol*, the consequences are undefined if a subsequent attempt is made to alter that *string*.

Examples:

```
(setq temp-string "temp") → "temp"
(setq temp-symbol (make-symbol temp-string)) → #:|temp|
(symbol-name temp-symbol) → "temp"
(eq (symbol-name temp-symbol) temp-string) → implementation-dependent
(find-symbol "temp") → NIL, NIL
(eq (make-symbol temp-string) (make-symbol temp-string)) → false
```

Exceptional Situations:

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

See Also:

copy-symbol

Notes:

No attempt is made by **make-symbol** to convert the case of the *name* to uppercase. The only case conversion which ever occurs for *symbols* is done by the *Lisp reader*. The program interface to *symbol* creation retains case, and the program interface to interning symbols is case-sensitive.

copy-symbol *Function*

Syntax:

copy-symbol *symbol* &optional *copy-properties* → *new-symbol*

Arguments and Values:

symbol—a *symbol*.

copy-properties—a *boolean*. The default is *false*.

new-symbol—a *fresh*, *uninterned symbol*.

Description:

copy-symbol returns a *fresh*, *uninterned symbol*, the *name* of which is **string=** to and possibly the *same* as the *name* of the given *symbol*.

If *copy-properties* is *false*, the *new-symbol* is neither *bound* nor *fbound* and has a *null property list*. If *copy-properties* is *true*, then the initial *value* of *new-symbol* is the *value* of *symbol*, the initial *function* definition of *new-symbol* is the *functional value* of *symbol*, and the *property list* of *new-symbol* is a *copy*$_2$ of the *property list* of *symbol*.

Examples:

```
(setq fred 'fred-smith) → FRED-SMITH
(setf (symbol-value fred) 3) → 3
(setq fred-clone-1a (copy-symbol fred nil)) → #:FRED-SMITH
(setq fred-clone-1b (copy-symbol fred nil)) → #:FRED-SMITH
(setq fred-clone-2a (copy-symbol fred t))   → #:FRED-SMITH
(setq fred-clone-2b (copy-symbol fred t))   → #:FRED-SMITH
(eq fred fred-clone-1a) → false
(eq fred-clone-1a fred-clone-1b) → false
(eq fred-clone-2a fred-clone-2b) → false
(eq fred-clone-1a fred-clone-2a) → false
(symbol-value fred) → 3
(boundp fred-clone-1a) → false
(symbol-value fred-clone-2a) → 3
(setf (symbol-value fred-clone-2a) 4) → 4
(symbol-value fred) → 3
(symbol-value fred-clone-2a) → 4
(symbol-value fred-clone-2b) → 3
(boundp fred-clone-1a) → false
(setf (symbol-function fred) #'(lambda (x) x)) → #<FUNCTION anonymous>
(fboundp fred) → true
(fboundp fred-clone-1a) → false
(fboundp fred-clone-2a) → false
```

Exceptional Situations:

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

See Also:

make-symbol

Notes:

Implementors are encouraged not to copy the *string* which is the *symbol*'s *name* unnecessarily. Unless there is a good reason to do so, the normal implementation strategy is for the *new-symbol*'s *name* to be *identical* to the given *symbol*'s *name*.

gensym *Function*

Syntax:

gensym &optional *x* → *new-symbol*

Arguments and Values:

x—a *string* or a non-negative *integer*. Complicated defaulting behavior; see below.

new-symbol—a *fresh*, *uninterned symbol*.

Description:

Creates and returns a *fresh*, *uninterned symbol*, as if by calling **make-symbol**. (The only difference between **gensym** and **make-symbol** is in how the *new-symbol*'s *name* is determined.)

The *name* of the *new-symbol* is the concatenation of a prefix, which defaults to "G", and a suffix, which is the decimal representation of a number that defaults to the *value* of ***gensym-counter***.

If *x* is supplied, and is a *string*, then that *string* is used as a prefix instead of "G" for this call to **gensym** only.

If *x* is supplied, and is an *integer*, then that *integer*, instead of the *value* of ***gensym-counter***, is used as the suffix for this call to **gensym** only.

If and only if no explicit suffix is supplied, ***gensym-counter*** is incremented after it is used.

Examples:

```
(setq sym1 (gensym)) → #:G3142
(symbol-package sym1) → NIL
(setq sym2 (gensym 100)) → #:G100
(setq sym3 (gensym 100)) → #:G100
(eq sym2 sym3) → false
(find-symbol "G100") → NIL, NIL
```

```
(gensym "T") → #:T3143
(gensym) → #:G3144
```

Side Effects:

Might increment ***gensym-counter***.

Affected By:

gensym-counter

Exceptional Situations:

Should signal an error of *type* **type-error** if *x* is not a *string* or a non-negative *integer*.

See Also:

gentemp, ***gensym-counter***

Notes:

The ability to pass a numeric argument to **gensym** has been deprecated; explicitly *binding* ***gensym-counter*** is now stylistically preferred. (The somewhat baroque conventions for the optional argument are historical in nature, and supported primarily for compatibility with older dialects of Lisp. In modern code, it is recommended that the only kind of argument used be a string prefix. In general, though, to obtain more flexible control of the *new-symbol*'s *name*, consider using **make-symbol** instead.)

gensym-counter — *Variable*

Value Type:

a non-negative *integer*.

Initial Value:

implementation-dependent.

Description:

A number which will be used in constructing the *name* of the next *symbol* generated by the *function* **gensym**.

gensym-counter can be either *assigned* or *bound* at any time, but its value must always be a non-negative *integer*.

Affected By:

gensym.

See Also:

gensym

Notes:

The ability to pass a numeric argument to **gensym** has been deprecated; explicitly *binding* ***gensym-counter*** is now stylistically preferred.

gentemp *Function*

Syntax:

gentemp &optional *prefix package* → *new-symbol*

Arguments and Values:

prefix—a *string*. The default is "T".

package—a *package designator*. The default is the *current package*.

new-symbol—a *fresh*, *interned symbol*.

Description:

gentemp creates and returns a *fresh symbol*, *interned* in the indicated *package*. The *symbol* is guaranteed to be one that was not previously *accessible* in *package*. It is neither *bound* nor *fbound*, and has a *null property list*.

The *name* of the *new-symbol* is the concatenation of the *prefix* and a suffix, which is taken from an internal counter used only by **gentemp**. (If a *symbol* by that name is already *accessible* in *package*, the counter is incremented as many times as is necessary to produce a *name* that is not already the *name* of a *symbol accessible* in *package*.)

Examples:

```
(gentemp) → T1298
(gentemp "FOO") → FOO1299
(find-symbol "FOO1300") → NIL, NIL
(gentemp "FOO") → FOO1300
(find-symbol "FOO1300") → FOO1300, :INTERNAL
(intern "FOO1301") → FOO1301, :INTERNAL
(gentemp "FOO") → FOO1302
(gentemp) → T1303
```

Side Effects:

Its internal counter is incremented one or more times.

Interns the *new-symbol* in *package*.

Affected By:

The current state of its internal counter, and the current state of the *package*.

Exceptional Situations:

Should signal an error of *type* **type-error** if *prefix* is not a *string*. Should signal an error of *type* **type-error** if *package* is not a *package designator*.

See Also:

gensym

Notes:

If *package* is the KEYWORD *package*, the result is an *external symbol* of *package*. Otherwise, the result is an *internal symbol* of *package*.

The **gentemp** internal counter is independent of ***gensym-counter***, the counter used by **gensym**. There is no provision for accessing the **gentemp** internal counter.

Just because **gentemp** creates a *symbol* which did not previously exist does not mean that such a *symbol* might not be seen in the future (*e.g.*, in a data file—perhaps even created by the same program in another session). As such, this symbol is not truly unique in the same sense as a *gensym* would be. In particular, programs which do automatic code generation should be careful not to attach global attributes to such generated *symbols* (*e.g.*, **special** *declarations*) and then write them into a file because such global attributes might, in a different session, end up applying to other *symbols* that were automatically generated on another day for some other purpose.

symbol-function *Accessor*

Syntax:

symbol-function *symbol* → *contents*

(**setf** (**symbol-function** *symbol*) *new-contents*)

Arguments and Values:

symbol—a *symbol*.

contents— If the *symbol* is globally defined as a *macro* or a *special operator*, an *object* of *implementation-dependent* nature and identity is returned. If the *symbol* is not globally defined as either a *macro* or a *special operator*, and if the *symbol* is *fbound*, a *function object* is returned.

new-contents—a *function*.

Description:

Accesses the *symbol*'s *function cell*.

symbol-function

Examples:

```
 (symbol-function 'car) → #<FUNCTION CAR>
 (symbol-function 'twice) is an error   ;because TWICE isn't defined.
 (defun twice (n) (* n 2)) → TWICE
 (symbol-function 'twice) → #<FUNCTION TWICE>
 (list (twice 3)
       (funcall (function twice) 3)
       (funcall (symbol-function 'twice) 3))
→ (6 6 6)
 (flet ((twice (x) (list x x)))
   (list (twice 3)
         (funcall (function twice) 3)
         (funcall (symbol-function 'twice) 3)))
→ ((3 3) (3 3) 6)
 (setf (symbol-function 'twice) #'(lambda (x) (list x x)))
→ #<FUNCTION anonymous>
 (list (twice 3)
       (funcall (function twice) 3)
       (funcall (symbol-function 'twice) 3))
→ ((3 3) (3 3) (3 3))
 (fboundp 'defun) → implementation-dependent
 (symbol-function 'defun) is an error   ;unless `guarded' by FBOUNDP
 (if (fboundp 'defun) (symbol-function 'defun))
→ implementation-dependent
 (if (fboundp 'defun) (functionp (symbol-function 'defun)))
→ implementation-dependent
 (defun symbol-function-or-nil (symbol)
   (if (and (fboundp symbol)
            (not (macro-function symbol))
            (not (special-operator-p symbol)))
       (symbol-function symbol)
       nil)) → SYMBOL-FUNCTION-OR-NIL
 (symbol-function-or-nil 'car) → #<FUNCTION CAR>
 (symbol-function-or-nil 'defun) → NIL
```

Affected By:

defun

Exceptional Situations:

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

Should signal **undefined-function** if *symbol* is not *fbound* and an attempt is made to *read* its definition. (No such error is signaled on an attempt to *write* its definition.)

See Also:

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

Notes:

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

setf may be used with **symbol-function** to replace a global function definition when the *symbol*'s function definition does not represent a *special operator*.

`(symbol-function` *symbol*`)` ≡ `(fdefinition` *symbol*`)`

However, **fdefinition** accepts arguments other than just *symbols*.

symbol-name *Function*

Syntax:

symbol-name *symbol* → *name*

Arguments and Values:

symbol—a *symbol*.

name—a *string*.

Description:

symbol-name returns the *name* of *symbol*. The consequences are undefined if *name* is ever modified.

Examples:

```
(symbol-name 'temp) → "TEMP"
(symbol-name :start) → "START"
(symbol-name (gensym)) → "G1234" ;for example
```

Exceptional Situations:

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

symbol-package

symbol-package *Function*

Syntax:

symbol-package *symbol* → *contents*

Arguments and Values:

symbol—a *symbol*.

contents—a *package object* or **nil**.

Description:

Returns the *home package* of *symbol*.

Examples:

```
(in-package "CL-USER") → #<PACKAGE "COMMON-LISP-USER">
(symbol-package 'car) → #<PACKAGE "COMMON-LISP">
(symbol-package 'bus) → #<PACKAGE "COMMON-LISP-USER">
(symbol-package :optional) → #<PACKAGE "KEYWORD">
;; Gensyms are uninterned, so have no home package.
(symbol-package (gensym)) → NIL
(make-package 'pk1) → #<PACKAGE "PK1">
(intern "SAMPLE1" "PK1") → PK1::SAMPLE1, NIL
(export (find-symbol "SAMPLE1" "PK1") "PK1") → T
(make-package 'pk2 :use '(pk1)) → #<PACKAGE "PK2">
(find-symbol "SAMPLE1" "PK2") → PK1:SAMPLE1, :INHERITED
(symbol-package 'pk1::sample1) → #<PACKAGE "PK1">
(symbol-package 'pk2::sample1) → #<PACKAGE "PK1">
(symbol-package 'pk1::sample2) → #<PACKAGE "PK1">
(symbol-package 'pk2::sample2) → #<PACKAGE "PK2">
;; The next several forms create a scenario in which a symbol
;; is not really uninterned, but is "apparently uninterned",
;; and so SYMBOL-PACKAGE still returns NIL.
(setq s3 'pk1::sample3) → PK1::SAMPLE3
(import s3 'pk2) → T
(unintern s3 'pk1) → T
(symbol-package s3) → NIL
(eq s3 'pk2::sample3) → T
```

Affected By:

import, **intern**, **unintern**

Exceptional Situations:

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

See Also:

intern

symbol-plist *Accessor*

Syntax:

symbol-plist *symbol* → *plist*

(**setf** (**symbol-plist** *symbol*) *new-plist*)

Arguments and Values:

symbol—a *symbol*.

plist, *new-plist*—a *property list*.

Description:

Accesses the *property list* of *symbol*.

Examples:

```
(setq sym (gensym)) → #:G9723
(symbol-plist sym) → ()
(setf (get sym 'prop1) 'val1) → VAL1
(symbol-plist sym) → (PROP1 VAL1)
(setf (get sym 'prop2) 'val2) → VAL2
(symbol-plist sym) → (PROP2 VAL2 PROP1 VAL1)
(setf (symbol-plist sym) (list 'prop3 'val3)) → (PROP3 VAL3)
(symbol-plist sym) → (PROP3 VAL3)
```

Exceptional Situations:

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

See Also:

get, **remprop**

Notes:

The use of **setf** should be avoided, since a *symbol*'s *property list* is a global resource that can contain information established and depended upon by unrelated programs in the same *Lisp image*.

symbol-value

symbol-value *Accessor*

Syntax:

symbol-value *symbol* → *value*

(**setf** (**symbol-value** *symbol*) *new-value*)

Arguments and Values:

symbol—a *symbol* that must have a *value*.

value, *new-value*—an *object*.

Description:

Accesses the *symbol*'s *value cell*.

Examples:

```
(setf (symbol-value 'a) 1) → 1
(symbol-value 'a) → 1
;; SYMBOL-VALUE cannot see lexical variables.
(let ((a 2)) (symbol-value 'a)) → 1
(let ((a 2)) (setq a 3) (symbol-value 'a)) → 1
;; SYMBOL-VALUE can see dynamic variables.
(let ((a 2))
  (declare (special a))
  (symbol-value 'a)) → 2
(let ((a 2))
  (declare (special a))
  (setq a 3)
  (symbol-value 'a)) → 3
(let ((a 2))
  (setf (symbol-value 'a) 3)
  a) → 2
a → 3
(symbol-value 'a) → 3
(let ((a 4))
  (declare (special a))
  (let ((b (symbol-value 'a)))
    (setf (symbol-value 'a) 5)
    (values a b))) → 5, 4
a → 3
(symbol-value :any-keyword) → :ANY-KEYWORD
(symbol-value 'nil) → NIL
(symbol-value '()) → NIL
```

```
;; The precision of this next one is implementation-dependent.
(symbol-value 'pi) → 3.141592653589793d0
```

Affected By:

makunbound, **set**, **setq**

Exceptional Situations:

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

Should signal **unbound-variable** if *symbol* is *unbound* and an attempt is made to *read* its *value*. (No such error is signaled on an attempt to *write* its *value*.)

See Also:

boundp, **makunbound**, **set**, **setq**

Notes:

symbol-value can be used to get the value of a *constant variable*. **symbol-value** cannot *access* the value of a *lexical variable*.

get *Accessor*

Syntax:

get *symbol indicator* `&optional` *default* → *value*

(**setf** (**get** *symbol indicator* `&optional` *default*) *new-value*)

Arguments and Values:

symbol—a *symbol*.

indicator—an *object*.

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

value—if the indicated property exists, the *object* that is its *value*; otherwise, the specified *default*.

new-value—an *object*.

Description:

get returns the corresponding value of an indicator from the *property list* of *symbol* **eq** to *indicator*, if one is found. Otherwise *default* is returned.

setf of **get** may be used to associate a new *object* with an existing indicator already on the *symbol*'s *property list*, or to create a new assocation. When a **get** *form* is used as a **setf** *place*,

get

any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its *value* is ignored.

Examples:

```
(defun make-person (first-name last-name)
  (let ((person (gensym "PERSON")))
    (setf (get person 'first-name) first-name)
    (setf (get person 'last-name) last-name)
    person)) → MAKE-PERSON
(defvar *john* (make-person "John" "Dow")) → *JOHN*
*john* → #:PERSON4603
(defvar *sally* (make-person "Sally" "Jones")) → *SALLY*
(get *john* 'first-name) → "John"
(get *sally* 'last-name) → "Jones"
(defun marry (man woman married-name)
  (setf (get man 'wife) woman)
  (setf (get woman 'husband) man)
  (setf (get man 'last-name) married-name)
  (setf (get woman 'last-name) married-name)
  married-name) → MARRY
(marry *john* *sally* "Dow-Jones") → "Dow-Jones"
(get *john* 'last-name) → "Dow-Jones"
(get (get *john* 'wife) 'first-name) → "Sally"
(symbol-plist *john*)
→ (WIFE #:PERSON4604 LAST-NAME "Dow-Jones" FIRST-NAME "John")
(defmacro age (person &optional (default ''thirty-something))
  `(get ,person 'age ,default)) → AGE
(age *john*) → THIRTY-SOMETHING
(age *john* 20) → 20
(setf (age *john*) 25) → 25
(age *john*) → 25
(age *john* 20) → 25
```

Exceptional Situations:

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

See Also:

getf, **symbol-plist**, **remprop**

Notes:

```
(get x y) ≡ (getf (symbol-plist x) y)
```

Numbers and *characters* are not recommended for use as *indicators* in portable code since **get** tests with **eq** rather than **eql**, and consequently the effect of using such *indicators* is *implementation-dependent*.

There is no way using **get** to distinguish an absent property from one whose value is *default*. However, see **get-properties**.

remprop *Function*

Syntax:

remprop *symbol indicator* → *boolean*

Arguments and Values:

symbol—a *symbol*.

indicator—an *object*.

boolean—a *boolean*.

Description:

Removes an entry from the *property list* of *symbol*, returning *false* if no such property was found, or *true* if a property was found.

The property of *symbol* with an indicator **eq** to *indicator* is removed. The property indicator and the corresponding value are removed in an undefined order by destructively splicing the property list.

(remprop *x y*) ≡ (remf (symbol-plist *x*) *y*)

Examples:

```
 (setq test (make-symbol "PSEUDO-PI")) → #:PSEUDO-PI
 (symbol-plist test) → ()
 (setf (get test 'constant) t) → T
 (setf (get test 'approximation) 3.14) → 3.14
 (setf (get test 'error-range) 'noticeable) → NOTICEABLE
 (symbol-plist test)
→ (ERROR-RANGE NOTICEABLE APPROXIMATION 3.14 CONSTANT T)
 (setf (get test 'approximation) nil) → NIL
 (symbol-plist test)
→ (ERROR-RANGE NOTICEABLE APPROXIMATION NIL CONSTANT T)
 (get test 'approximation) → NIL
 (remprop test 'approximation) → true
 (get test 'approximation) → NIL
 (symbol-plist test)
→ (ERROR-RANGE NOTICEABLE CONSTANT T)
```

```
 (remprop test 'approximation) → NIL
 (symbol-plist test)
→ (ERROR-RANGE NOTICEABLE CONSTANT T)
 (remprop test 'error-range) → true
 (setf (get test 'approximation) 3) → 3
 (symbol-plist test)
→ (APPROXIMATION 3 CONSTANT T)
```

Side Effects:

The *property list* of *symbol* is modified.

Exceptional Situations:

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

See Also:

remf, **symbol-plist**

Notes:

Numbers and *characters* are not recommended for use as *indicators* in portable code since **remprop** tests with **eq** rather than **eql**, and consequently the effect of using such *indicators* is *implementation-dependent*. Of course, if you've gotten as far as needing to remove such a *property*, you don't have much choice—the time to have been thinking about this was when you used **setf** of **get** to establish the *property*.

boundp *Function*

Syntax:

boundp *symbol* → *boolean*

Arguments and Values:

symbol—a *symbol*.

boolean—a *boolean*.

Description:

Returns *true* if *symbol* is *bound*; otherwise, returns *false*.

Examples:

```
(setq x 1) → 1
(boundp 'x) → true
(makunbound 'x) → X
(boundp 'x) → false
```

```
(let ((x 2)) (boundp 'x)) → false
(let ((x 2)) (declare (special x)) (boundp 'x)) → true
```

Exceptional Situations:

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

See Also:

set, **setq**, **symbol-value**, **makunbound**

Notes:

The *function* **bound** determines only whether a *symbol* has a value in the *global environment*; any *lexical bindings* are ignored.

makunbound *Function*

Syntax:

makunbound *symbol* → *symbol*

Arguments and Values:

symbol—a *symbol*

Description:

Makes the *symbol* be *unbound*, regardless of whether it was previously *bound*.

Examples:

```
(setf (symbol-value 'a) 1)
(boundp 'a) → true
a → 1
(makunbound 'a) → A
(boundp 'a) → false
```

Side Effects:

The *value cell* of *symbol* is modified.

Exceptional Situations:

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

See Also:

boundp, **fmakunbound**

set

set *Function*

Syntax:

set *symbol value* → *value*

Arguments and Values:

symbol—a *symbol*.

value—an *object*.

Description:

set changes the contents of the *value cell* of *symbol* to the given *value*.

(set *symbol value*) ≡ (setf (symbol-value *symbol*) *value*)

Examples:

```
(setf (symbol-value 'n) 1) → 1
(set 'n 2) → 2
(symbol-value 'n) → 2
(let ((n 3))
  (declare (special n))
  (setq n (+ n 1))
  (setf (symbol-value 'n) (* n 10))
  (set 'n (+ (symbol-value 'n) n))
  n) → 80
n → 2
(let ((n 3))
  (setq n (+ n 1))
  (setf (symbol-value 'n) (* n 10))
  (set 'n (+ (symbol-value 'n) n))
  n) → 4
n → 44
(defvar *n* 2)
(let ((*n* 3))
  (setq *n* (+ *n* 1))
  (setf (symbol-value '*n*) (* *n* 10))
  (set '*n* (+ (symbol-value '*n*) *n*))
  *n*) → 80
 *n* → 2
(defvar *even-count* 0) → *EVEN-COUNT*
(defvar *odd-count* 0) → *ODD-COUNT*
(defun tally-list (list)
  (dolist (element list)
    (set (if (evenp element) '*even-count* '*odd-count*)
```

```
          (+ element (if (evenp element) *even-count* *odd-count*)))))
 (tally-list '(1 9 4 3 2 7)) → NIL
 *even-count* → 6
 *odd-count* → 20
```

Side Effects:

The *value* of *symbol* is changed.

See Also:

setq, **progv**, **symbol-value**

Notes:

set cannot change the value of a *lexical variable*.

unbound-variable *Condition Type*

Class Precedence List:

unbound-variable, **cell-error**, **error**, **serious-condition**, **condition**, **t**

Description:

The *type* **unbound-variable** consists of *error conditions* that represent attempts to *read* the *value* of an *unbound variable*.

The name of the cell (see **cell-error**) is the *name* of the *variable* that was *unbound*.

See Also:

cell-error-name

Table of Contents