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

Programming Language—Common Lisp

14. Conses

14.1 Cons Concepts

A ***cons*** is a compound data *object* having two components called the *car* and the *cdr*.

car	**cons**	**rplacd**
cdr	**rplaca**	

Figure 14–1. Some defined names relating to conses.

Depending on context, a group of connected *conses* can be viewed in a variety of different ways. A variety of operations is provided to support each of these various views.

14.1.1 Conses as Trees

A ***tree*** is a binary recursive data structure made up of *conses* and *atoms*: the *conses* are themselves also *trees* (sometimes called "subtrees" or "branches"), and the *atoms* are terminal nodes (sometimes called ***leaves***). Typically, the *leaves* represent data while the branches establish some relationship among that data.

caaaar	**caddar**	**cdar**	**nsubst**
caaadr	**cadddr**	**cddaar**	**nsubst-if**
caaar	**caddr**	**cddadr**	**nsubst-if-not**
caadar	**cadr**	**cddar**	**nthcdr**
caaddr	**cdaaar**	**cdddar**	**sublis**
caadr	**cdaadr**	**cddddr**	**subst**
caar	**cdaar**	**cdddr**	**subst-if**
cadaar	**cdadar**	**cddr**	**subst-if-not**
cadadr	**cdaddr**	**copy-tree**	**tree-equal**
cadar	**cdadr**	**nsublis**	

Figure 14–2. Some defined names relating to trees.

14.1.1.1 General Restrictions on Parameters that must be Trees

Except as explicitly stated otherwise, for any *standardized function* that takes a *parameter* that is required to be a *tree*, the consequences are undefined if that *tree* is circular.

14.1.2 Conses as Lists

A ***list*** is a chain of *conses* in which the *car* of each *cons* is an *element* of the *list*, and the *cdr* of each *cons* is either the next link in the chain or a terminating *atom*.

A ***proper list*** is a *list* terminated by the *empty list*. The *empty list* is a *proper list*, but is not a *cons*.

An ***improper list*** is a *list* that is not a *proper list*; that is, it is a *circular list* or a *dotted list*.

A ***dotted list*** is a *list* that has a terminating *atom* that is not the *empty list*. A *non-nil atom* by itself is not considered to be a *list* of any kind—not even a *dotted list*.

A ***circular list*** is a chain of *conses* that has no termination because some *cons* in the chain is the *cdr* of a later *cons*.

append	**last**	**nbutlast**	**rest**
butlast	**ldiff**	**nconc**	**revappend**
copy-alist	**list**	**ninth**	**second**
copy-list	**list***	**nreconc**	**seventh**
eighth	**list-length**	**nth**	**sixth**
endp	**make-list**	**nthcdr**	**tailp**
fifth	**member**	**pop**	**tenth**
first	**member-if**	**push**	**third**
fourth	**member-if-not**	**pushnew**	

Figure 14–3. Some defined names relating to lists.

14.1.2.1 Lists as Association Lists

An ***association list*** is a *list* of *conses* representing an association of *keys* with *values*, where the *car* of each *cons* is the *key* and the *cdr* is the *value* associated with that *key*.

acons	**assoc-if**	**pairlis**	**rassoc-if**
assoc	**assoc-if-not**	**rassoc**	**rassoc-if-not**

Figure 14–4. Some defined names related to assocation lists.

14.1.2.2 Lists as Sets

Lists are sometimes viewed as sets by considering their elements unordered and by assuming there is no duplication of elements.

adjoin	**nset-difference**	**set-difference**	**union**
intersection	**nset-exclusive-or**	**set-exclusive-or**	
nintersection	**nunion**	**subsetp**	

Figure 14–5. Some defined names related to sets.

14.1.2.3 General Restrictions on Parameters that must be Lists

Except as explicitly specified otherwise, any *standardized function* that takes a *parameter* that is required to be a *list* should be prepared to signal an error of *type* **type-error** if the *value* received is a *dotted list*.

Except as explicitly specified otherwise, for any *standardized function* that takes a *parameter* that is required to be a *list*, the consequences are undefined if that *list* is *circular*.

list *System Class*

Class Precedence List:

list, **sequence**, **t**

Description:

A ***list*** is a chain of *conses* in which the *car* of each *cons* is an *element* of the *list*, and the *cdr* of each *cons* is either the next link in the chain or a terminating *atom*.

A ***proper list*** is a chain of *conses* terminated by the ***empty list***, `()`, which is itself a *proper list*. A ***dotted list*** is a *list* which has a terminating *atom* that is not the *empty list*. A ***circular list*** is a chain of *conses* that has no termination because some *cons* in the chain is the *cdr* of a later *cons*.

Dotted lists and *circular lists* are also *lists*, but usually the unqualified term "list" within this specification means *proper list*. Nevertheless, the *type* **list** unambiguously includes *dotted lists* and *circular lists*.

For each *element* of a *list* there is a *cons*. The *empty list* has no *elements* and is not a *cons*.

The *types* **cons** and **null** form an *exhaustive partition* of the *type* **list**.

See Also:

Section 2.4.1 (Left-Parenthesis), Section 22.1.3.5 (Printing Lists and Conses)

null *System Class*

Class Precedence List:

null, **symbol**, **list**, **sequence**, **t**

Description:

The only *object* of *type* **null** is **nil**, which represents the *empty list* and can also be notated `()`.

See Also:

Section 2.3.4 (Symbols as Tokens), Section 2.4.1 (Left-Parenthesis), Section 22.1.3.3 (Printing Symbols)

cons *System Class*

Class Precedence List:

cons, **list**, **sequence**, **t**

Description:

A *cons* is a compound *object* having two components, called the *car* and *cdr*. These form a *dotted pair*. Each component can be any *object*.

Compound Type Specifier Kind:

Specializing.

Compound Type Specifier Syntax:

`(cons [`*car-typespec* `[`*cdr-typespec*`]])`

Compound Type Specifier Arguments:

car-typespec—a *type specifier*, or the *symbol* `*`. The default is the *symbol* `*`.

cdr-typespec—a *type specifier*, or the *symbol* `*`. The default is the *symbol* `*`.

Compound Type Specifier Description:

This denotes the set of *conses* whose *car* is constrained to be of *type* *car-typespec* and whose *cdr* is constrained to be of *type* *cdr-typespec*. (If either *car-typespec* or *cdr-typespec* is `*`, it is as if the *type* **t** had been denoted.)

See Also:

Section 2.4.1 (Left-Parenthesis), Section 22.1.3.5 (Printing Lists and Conses)

atom *Type*

Supertypes:

atom, **t**

Description:

It is equivalent to `(not cons)`.

cons *Function*

Syntax:

cons *object-1 object-2* → *cons*

Arguments and Values:

object-1—an *object*.

object-2—an *object*.

cons—a *cons*.

Description:

Creates a *fresh cons*, the *car* of which is *object-1* and the *cdr* of which is *object-2*.

Examples:

```
(cons 1 2) → (1 . 2)
(cons 1 nil) → (1)
(cons nil 2) → (NIL . 2)
(cons nil nil) → (NIL)
(cons 1 (cons 2 (cons 3 (cons 4 nil)))) → (1 2 3 4)
(cons 'a 'b) → (A . B)
(cons 'a (cons 'b (cons 'c '()))) → (A B C)
(cons 'a '(b c d)) → (A B C D)
```

See Also:

list

Notes:

If *object-2* is a *list*, **cons** can be thought of as producing a new *list* which is like it but has *object-1* prepended.

consp *Function*

Syntax:

consp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

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

Examples:

```
(consp nil) → false
(consp (cons 1 2)) → true
```

The *empty list* is not a *cons*, so

```
(consp '()) ≡ (consp 'nil) → false
```

See Also:

listp

Notes:

```
(consp object) ≡ (typep object 'cons) ≡ (not (typep object 'atom)) ≡ (typep object '(not atom))
```

atom *Function*

Syntax:

atom *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

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

Examples:

```
(atom 'sss) → true
(atom (cons 1 2)) → false
(atom nil) → true
(atom '()) → true
(atom 3) → true
```

Notes:

```
(atom object) ≡ (typep object 'atom) ≡ (not (consp object))
≡ (not (typep object 'cons)) ≡ (typep object '(not cons))
```

rplaca, rplacd *Function*

Syntax:

rplaca *cons object* → *cons*
rplacd *cons object* → *cons*

Pronunciation:

rplaca: [ˌrē ˈplakϵ] or [ˌrϵ ˈplakϵ]

rplacd: [ˌrē ˈplakdϵ] or [ˌrϵ ˈplakdϵ] or [ˌrē ˈplakdē] or [ˌrϵ ˈplakdē]

Arguments and Values:

cons—a *cons*.

object—an *object*.

Description:

rplaca replaces the *car* of the *cons* with *object*.

rplacd replaces the *cdr* of the *cons* with *object*.

Examples:

```
(defparameter *some-list* (list* 'one 'two 'three 'four)) → *some-list*
*some-list* → (ONE TWO THREE . FOUR)
(rplaca *some-list* 'uno) → (UNO TWO THREE . FOUR)
*some-list* → (UNO TWO THREE . FOUR)
(rplacd (last *some-list*) (list 'IV)) → (THREE IV)
*some-list* → (UNO TWO THREE IV)
```

Side Effects:

The *cons* is modified.

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

car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, caddr, cdaar, cdadr, cddar, cdddr, caaaar, caaadr, caadar, caaddr, cadaar, cadadr, caddar, cadddr, cdaaar, cdaadr, cdadar, cdaddr, cddaar, cddadr, cdddar, cddddr *Accessor*

Syntax:

car *x*	→ *object*	(**setf** (**car** *x*) *new-object*)
cdr *x*	→ *object*	(**setf** (**cdr** *x*) *new-object*)
caar *x*	→ *object*	(**setf** (**caar** *x*) *new-object*)
cadr *x*	→ *object*	(**setf** (**cadr** *x*) *new-object*)
cdar *x*	→ *object*	(**setf** (**cdar** *x*) *new-object*)
cddr *x*	→ *object*	(**setf** (**cddr** *x*) *new-object*)
caaar *x*	→ *object*	(**setf** (**caaar** *x*) *new-object*)
caadr *x*	→ *object*	(**setf** (**caadr** *x*) *new-object*)
cadar *x*	→ *object*	(**setf** (**cadar** *x*) *new-object*)
caddr *x*	→ *object*	(**setf** (**caddr** *x*) *new-object*)
cdaar *x*	→ *object*	(**setf** (**cdaar** *x*) *new-object*)
cdadr *x*	→ *object*	(**setf** (**cdadr** *x*) *new-object*)
cddar *x*	→ *object*	(**setf** (**cddar** *x*) *new-object*)
cdddr *x*	→ *object*	(**setf** (**cdddr** *x*) *new-object*)
caaaar *x*	→ *object*	(**setf** (**caaaar** *x*) *new-object*)
caaadr *x*	→ *object*	(**setf** (**caaadr** *x*) *new-object*)
caadar *x*	→ *object*	(**setf** (**caadar** *x*) *new-object*)
caaddr *x*	→ *object*	(**setf** (**caaddr** *x*) *new-object*)
cadaar *x*	→ *object*	(**setf** (**cadaar** *x*) *new-object*)
cadadr *x*	→ *object*	(**setf** (**cadadr** *x*) *new-object*)
caddar *x*	→ *object*	(**setf** (**caddar** *x*) *new-object*)
cadddr *x*	→ *object*	(**setf** (**cadddr** *x*) *new-object*)
cdaaar *x*	→ *object*	(**setf** (**cdaaar** *x*) *new-object*)
cdaadr *x*	→ *object*	(**setf** (**cdaadr** *x*) *new-object*)
cdadar *x*	→ *object*	(**setf** (**cdadar** *x*) *new-object*)
cdaddr *x*	→ *object*	(**setf** (**cdaddr** *x*) *new-object*)
cddaar *x*	→ *object*	(**setf** (**cddaar** *x*) *new-object*)
cddadr *x*	→ *object*	(**setf** (**cddadr** *x*) *new-object*)
cdddar *x*	→ *object*	(**setf** (**cdddar** *x*) *new-object*)
cddddr *x*	→ *object*	(**setf** (**cddddr** *x*) *new-object*)

Pronunciation:

cadr: [ˈ**ka**ˌ**d**ɛ**r**]

car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...

caddr: [ˈ**kadə͵dər**] or [ˈ**ka͵dủdər**]

cdr: [ˈ**kủ͵dər**]

cddr: [ˈ**kủdə͵dər**] or [ˈ**kə͵dủdər**]

Arguments and Values:

x—a *list*.

object—an *object*.

new-object—an *object*.

Description:

If *x* is a *cons*, **car** returns the *car* of that *cons*. If *x* is **nil**, **car** returns **nil**.

If *x* is a *cons*, **cdr** returns the *cdr* of that *cons*. If *x* is **nil**, **cdr** returns **nil**.

Functions are provided which perform compositions of up to four **car** and **cdr** operations. Their *names* consist of a `C`, followed by two, three, or four occurrences of `A` or `D`, and finally an `R`. The series of `A`'s and `D`'s in each *function*'s *name* is chosen to identify the series of **car** and **cdr** operations that is performed by the function. The order in which the `A`'s and `D`'s appear is the inverse of the order in which the corresponding operations are performed. Figure 14–6 defines the relationships precisely.

car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, ...

This *place* ...	**Is equivalent to this** *place* ...
(caar *x*)	(car (car *x*))
(cadr *x*)	(car (cdr *x*))
(cdar *x*)	(cdr (car *x*))
(cddr *x*)	(cdr (cdr *x*))
(caaar *x*)	(car (car (car *x*)))
(caadr *x*)	(car (car (cdr *x*)))
(cadar *x*)	(car (cdr (car *x*)))
(caddr *x*)	(car (cdr (cdr *x*)))
(cdaar *x*)	(cdr (car (car *x*)))
(cdadr *x*)	(cdr (car (cdr *x*)))
(cddar *x*)	(cdr (cdr (car *x*)))
(cdddr *x*)	(cdr (cdr (cdr *x*)))
(caaaar *x*)	(car (car (car (car *x*))))
(caaadr *x*)	(car (car (car (cdr *x*))))
(caadar *x*)	(car (car (cdr (car *x*))))
(caaddr *x*)	(car (car (cdr (cdr *x*))))
(cadaar *x*)	(car (cdr (car (car *x*))))
(cadadr *x*)	(car (cdr (car (cdr *x*))))
(caddar *x*)	(car (cdr (cdr (car *x*))))
(cadddr *x*)	(car (cdr (cdr (cdr *x*))))
(cdaaar *x*)	(cdr (car (car (car *x*))))
(cdaadr *x*)	(cdr (car (car (cdr *x*))))
(cdadar *x*)	(cdr (car (cdr (car *x*))))
(cdaddr *x*)	(cdr (car (cdr (cdr *x*))))
(cddaar *x*)	(cdr (cdr (car (car *x*))))
(cddadr *x*)	(cdr (cdr (car (cdr *x*))))
(cdddar *x*)	(cdr (cdr (cdr (car *x*))))
(cddddr *x*)	(cdr (cdr (cdr (cdr *x*))))

Figure 14–6. CAR and CDR variants

setf can also be used with any of these functions to change an existing component of *x*, but **setf** will not make new components. So, for example, the *car* of a *cons* can be assigned with **setf** of **car**, but the *car* of **nil** cannot be assigned with **setf** of **car**. Similarly, the *car* of the *car* of a *cons* whose *car* is a *cons* can be assigned with **setf** of **caar**, but neither **nil**nor a *cons* whose car is **nil** can be assigned with **setf** of **caar**.

The argument *x* is permitted to be a *dotted list* or a *circular list*.

Examples:

```
(car nil) → NIL
(cdr '(1 . 2)) → 2
(cdr '(1 2)) → (2)
```

```
(cadr '(1 2)) → 2
(car '(a b c)) → A
(cdr '(a b c)) → (B C)
```

Exceptional Situations:

The functions **car** and **cdr** should signal **type-error** if they receive an argument which is not a *list*. The other functions (**caar**, **cadr**, ... **cddddr**) should behave for the purpose of error checking as if defined by appropriate calls to **car** and **cdr**.

See Also:

rplaca, **first**, **rest**

Notes:

The *car* of a *cons* can also be altered by using **rplaca**, and the *cdr* of a *cons* can be altered by using **rplacd**.

```
(car x)     ≡ (first x)
(cadr x)    ≡ (second x) ≡ (car (cdr x))
(caddr x)   ≡ (third x)  ≡ (car (cdr (cdr x)))
(cadddr x)  ≡ (fourth x) ≡ (car (cdr (cdr (cdr x))))
```

copy-tree *Function*

Syntax:

copy-tree *tree* → *new-tree*

Arguments and Values:

tree—a *tree*.

new-tree—a *tree*.

Description:

Creates a *copy* of a *tree* of *conses*.

If *tree* is not a *cons*, it is returned; otherwise, the result is a new *cons* of the results of calling **copy-tree** on the *car* and *cdr* of *tree*. In other words, all *conses* in the *tree* represented by *tree* are copied recursively, stopping only when non-*conses* are encountered.

copy-tree does not preserve circularities and the sharing of substructure.

Examples:

```
(setq object (list (cons 1 "one")
```

```
                    (cons 2 (list 'a 'b 'c))))
→ ((1 . "one") (2 A B C))
 (setq object-too object) → ((1 . "one") (2 A B C))
 (setq copy-as-list (copy-list object))
 (setq copy-as-alist (copy-alist object))
 (setq copy-as-tree (copy-tree object))
 (eq object object-too) → true
 (eq copy-as-tree object) → false
 (eql copy-as-tree object) → false
 (equal copy-as-tree object) → true
 (setf (first (cdr (second object))) "a"
       (car (second object)) "two"
       (car object) '(one . 1)) → (ONE . 1)
 object → ((ONE . 1) ("two" "a" B C))
 object-too → ((ONE . 1) ("two" "a" B C))
 copy-as-list → ((1 . "one") ("two" "a" B C))
 copy-as-alist → ((1 . "one") (2 "a" B C))
 copy-as-tree → ((1 . "one") (2 A B C))
```

See Also:

tree-equal

sublis, nsublis *Function*

Syntax:

sublis *alist tree* &key *key test test-not* → *new-tree*

nsublis *alist tree* &key *key test test-not* → *new-tree*

Arguments and Values:

alist—an *association list*.

tree—a *tree*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

new-tree—a *tree*.

sublis, nsublis

Description:

sublis makes substitutions for *objects* in *tree* (a structure of *conses*). **nsublis** is like **sublis** but destructively modifies the relevant parts of the *tree*.

sublis looks at all subtrees and leaves of *tree*; if a subtree or leaf appears as a key in *alist* (that is, the key and the subtree or leaf *satisfy the test*), it is replaced by the *object* with which that key is associated. This operation is non-destructive. In effect, **sublis** can perform several **subst** operations simultaneously.

If **sublis** succeeds, a new copy of *tree* is returned in which each occurrence of such a subtree or leaf is replaced by the *object* with which it is associated. If no changes are made, the original tree is returned. The original *tree* is left unchanged, but the result tree may share cells with it.

nsublis is permitted to modify *tree* but otherwise returns the same values as **sublis**.

Examples:

```
 (sublis '((x . 100) (z . zprime))
         '(plus x (minus g z x p) 4 . x))
→ (PLUS 100 (MINUS G ZPRIME 100 P) 4 . 100)
 (sublis '(((+ x y) . (- x y)) ((- x y) . (+ x y)))
         '(* (/ (+ x y) (+ x p)) (- x y))
         :test #'equal)
→ (* (/ (- X Y) (+ X P)) (+ X Y))
 (setq tree1 '(1 (1 2) ((1 2 3)) (((1 2 3 4)))))
→ (1 (1 2) ((1 2 3)) (((1 2 3 4))))
 (sublis '((3 . "three")) tree1)
→ (1 (1 2) ((1 2 "three")) (((1 2 "three" 4))))
 (sublis '((t . "string"))
          (sublis '((1 . "") (4 . 44)) tree1)
          :key #'stringp)
→ ("string" ("string" 2) (("string" 2 3)) ((("string" 2 3 44))))
 tree1 → (1 (1 2) ((1 2 3)) (((1 2 3 4))))
 (setq tree2 '("one" ("one" "two") (("one" "Two" "three"))))
→ ("one" ("one" "two") (("one" "Two" "three")))
 (sublis '(("two" . 2)) tree2)
→ ("one" ("one" "two") (("one" "Two" "three")))
 tree2 → ("one" ("one" "two") (("one" "Two" "three")))
 (sublis '(("two" . 2)) tree2 :test 'equal)
→ ("one" ("one" 2) (("one" "Two" "three")))

 (nsublis '((t . 'temp))
           tree1
           :key #'(lambda (x) (or (atom x) (< (list-length x) 3))))
→ ((QUOTE TEMP) (QUOTE TEMP) QUOTE TEMP)
```

Side Effects:

nsublis modifies *tree*.

See Also:

subst, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

Because the side-effecting variants (*e.g.*, **nsublis**) potentially change the path that is being traversed, their effects in the presence of shared or circular structure structure may vary in surprising ways when compared to their non-side-effecting alternatives. To see this, consider the following side-effect behavior, which might be exhibited by some implementations:

```
(defun test-it (fn)
  (let* ((shared-piece (list 'a 'b))
         (data (list shared-piece shared-piece)))
    (funcall fn '((a . b) (b . a)) data)))
(test-it #'sublis) → ((B A) (B A))
(test-it #'nsublis) → ((A B) (A B))
```

subst, subst-if, subst-if-not, nsubst, nsubst-if, nsubst-if-not *Function*

Syntax:

subst *new old tree* `&key` *key test test-not* → *new-tree*

subst-if *new predicate tree* `&key` *key* → *new-tree*

subst-if-not *new predicate tree* `&key` *key* → *new-tree*

nsubst *new old tree* `&key` *key test test-not* → *new-tree*

nsubst-if *new predicate tree* `&key` *key* → *new-tree*

nsubst-if-not *new predicate tree* `&key` *key* → *new-tree*

Arguments and Values:

new—an *object*.

old—an *object*.

subst, subst-if, subst-if-not, nsubst, nsubst-if, ...

predicate—a *symbol* that names a *function*, or a *function* of one argument that returns a *generalized boolean* value.

tree—a *tree*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

new-tree—a *tree*.

Description:

subst, **subst-if**, and **subst-if-not** perform substitution operations on *tree*. Each function searches *tree* for occurrences of a particular *old* item of an element or subexpression that *satisfies the test*.

nsubst, **nsubst-if**, and **nsubst-if-not** are like **subst**, **subst-if**, and **subst-if-not** respectively, except that the original *tree* is modified.

subst makes a copy of *tree*, substituting *new* for every subtree or leaf of *tree* (whether the subtree or leaf is a *car* or a *cdr* of its parent) such that *old* and the subtree or leaf *satisfy the test*.

nsubst is a destructive version of **subst**. The list structure of *tree* is altered by destructively replacing with *new* each leaf of the *tree* such that *old* and the leaf *satisfy the test*.

For **subst**, **subst-if**, and **subst-if-not**, if the functions succeed, a new copy of the tree is returned in which each occurrence of such an element is replaced by the *new* element or subexpression. If no changes are made, the original *tree* may be returned. The original *tree* is left unchanged, but the result tree may share storage with it.

For **nsubst**, **nsubst-if**, and **nsubst-if-not** the original *tree* is modified and returned as the function result, but the result may not be **eq** to *tree*.

Examples:

```
 (setq tree1 '(1 (1 2) (1 2 3) (1 2 3 4))) → (1 (1 2) (1 2 3) (1 2 3 4))
 (subst "two" 2 tree1) → (1 (1 "two") (1 "two" 3) (1 "two" 3 4))
 (subst "five" 5 tree1) → (1 (1 2) (1 2 3) (1 2 3 4))
 (eq tree1 (subst "five" 5 tree1)) → implementation-dependent
 (subst 'tempest 'hurricane
        '(shakespeare wrote (the hurricane)))
→ (SHAKESPEARE WROTE (THE TEMPEST))
 (subst 'foo 'nil '(shakespeare wrote (twelfth night)))
→ (SHAKESPEARE WROTE (TWELFTH NIGHT . FOO) . FOO)
 (subst '(a . cons) '(old . pair)
        '((old . spice) ((old . shoes) old . pair) (old . pair))
```

```
        :test #'equal)
→ ((OLD . SPICE) ((OLD . SHOES) A . CONS) (A . CONS))

 (subst-if 5 #'listp tree1) → 5
 (subst-if-not '(x) #'consp tree1)
→ (1 X)

 tree1 → (1 (1 2) (1 2 3) (1 2 3 4))
 (nsubst 'x 3 tree1 :key #'(lambda (y) (and (listp y) (third y))))
→ (1 (1 2) X X)
 tree1 → (1 (1 2) X X)
```

Side Effects:

nsubst, **nsubst-if**, and **nsubst-if-not** might alter the *tree structure* of *tree*.

See Also:

substitute, **nsubstitute**, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

The functions **subst-if-not** and **nsubst-if-not** are deprecated.

One possible definition of **subst**:

```
 (defun subst (old new tree &rest x &key test test-not key)
   (cond ((satisfies-the-test old tree :test test
                              :test-not test-not :key key)
          new)
         ((atom tree) tree)
         (t (let ((a (apply #'subst old new (car tree) x))
                  (d (apply #'subst old new (cdr tree) x)))
              (if (and (eql a (car tree))
                       (eql d (cdr tree)))
                  tree
                  (cons a d))))))
```

tree-equal

tree-equal *Function*

Syntax:

tree-equal *tree-1 tree-2* &key *test test-not* → *generalized-boolean*

Arguments and Values:

tree-1—a *tree*.

tree-2—a *tree*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

generalized-boolean—a *generalized boolean*.

Description:

tree-equal tests whether two trees are of the same shape and have the same leaves. **tree-equal** returns *true* if *tree-1* and *tree-2* are both *atoms* and *satisfy the test*, or if they are both *conses* and the *car* of *tree-1* is **tree-equal** to the *car* of *tree-2* and the *cdr* of *tree-1* is **tree-equal** to the *cdr* of *tree-2*. Otherwise, **tree-equal** returns *false*.

tree-equal recursively compares *conses* but not any other *objects* that have components.

The first argument to the `:test` or `:test-not` function is *tree-1* or a *car* or *cdr* of *tree-1*; the second argument is *tree-2* or a *car* or *cdr* of *tree-2*.

Examples:

```
 (setq tree1 '(1 (1 2))
       tree2 '(1 (1 2))) → (1 (1 2))
 (tree-equal tree1 tree2) → true
 (eql tree1 tree2) → false
 (setq tree1 '('a ('b 'c))
       tree2 '('a ('b 'c))) → ('a ('b 'c))
→ ((QUOTE A) ((QUOTE B) (QUOTE C)))
 (tree-equal tree1 tree2 :test 'eq) → true
```

Exceptional Situations:

The consequences are undefined if both *tree-1* and *tree-2* are circular.

See Also:

equal, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

copy-list *Function*

Syntax:

copy-list *list* → *copy*

Arguments and Values:

list—a *proper list* or a *dotted list*.

copy—a *list*.

Description:

Returns a *copy* of *list*. If *list* is a *dotted list*, the resulting *list* will also be a *dotted list*.

Only the *list structure* of *list* is copied; the *elements* of the resulting list are the *same* as the corresponding *elements* of the given *list*.

Examples:

```
(setq lst (list 1 (list 2 3))) → (1 (2 3))
(setq slst lst) → (1 (2 3))
(setq clst (copy-list lst)) → (1 (2 3))
(eq slst lst) → true
(eq clst lst) → false
(equal clst lst) → true
(rplaca lst "one") → ("one" (2 3))
slst → ("one" (2 3))
clst → (1 (2 3))
(setf (caadr lst) "two") → "two"
lst → ("one" ("two" 3))
slst → ("one" ("two" 3))
clst → (1 ("two" 3))
```

Exceptional Situations:

The consequences are undefined if *list* is a *circular list*.

See Also:

copy-alist, **copy-seq**, **copy-tree**

Notes:

The copy created is **equal** to *list*, but not **eq**.

list, list∗

list, list∗ *Function*

Syntax:

list &rest *objects* → *list*

list* &rest *objects*+ → *result*

Arguments and Values:

object—an *object*.

list—a *list*.

result—an *object*.

Description:

list returns a *list* containing the supplied *objects*.

list* is like **list** except that the last *argument* to **list** becomes the *car* of the last *cons* constructed, while the last *argument* to **list*** becomes the *cdr* of the last *cons* constructed. Hence, any given call to **list*** always produces one fewer *conses* than a call to **list** with the same number of arguments.

If the last *argument* to **list*** is a *list*, the effect is to construct a new *list* which is similar, but which has additional elements added to the front corresponding to the preceding *arguments* of **list***.

If **list*** receives only one *object*, that *object* is returned, regardless of whether or not it is a *list*.

Examples:

```
(list 1) → (1)
(list* 1) → 1
(setq a 1) → 1
(list a 2) → (1 2)
'(a 2) → (A 2)
(list 'a 2) → (A 2)
(list* a 2) → (1 . 2)
(list) → NIL ;i.e.,  ()
(setq a '(1 2)) → (1 2)
(eq a (list* a)) → true
(list 3 4 'a (car '(b . c)) (+ 6 -2)) → (3 4 A B 4)
(list* 'a 'b 'c 'd) ≡ (cons 'a (cons 'b (cons 'c 'd))) → (A B C . D)
(list* 'a 'b 'c '(d e f)) → (A B C D E F)
```

See Also:

cons

Notes:

```
(list* x) ≡ x
```

list-length *Function*

Syntax:

list-length *list* → *length*

Arguments and Values:

list—a *proper list* or a *circular list*.

length—a non-negative *integer*, or **nil**.

Description:

Returns the *length* of *list* if *list* is a *proper list*. Returns **nil** if *list* is a *circular list*.

Examples:

```
(list-length '(a b c d)) → 4
(list-length '(a (b c) d)) → 3
(list-length '()) → 0
(list-length nil) → 0
(defun circular-list (&rest elements)
  (let ((cycle (copy-list elements)))
    (nconc cycle cycle)))
(list-length (circular-list 'a 'b)) → NIL
(list-length (circular-list 'a)) → NIL
(list-length (circular-list)) → 0
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *list* is not a *proper list* or a *circular list*.

See Also:

length

Notes:

list-length could be implemented as follows:

```
(defun list-length (x)
  (do ((n 0 (+ n 2))           ;Counter.
       (fast x (cddr fast))    ;Fast pointer: leaps by 2.
```

```
     (slow x (cdr slow)))    ;Slow pointer: leaps by 1.
    (nil)
  ;; If fast pointer hits the end, return the count.
  (when (endp fast) (return n))
  (when (endp (cdr fast)) (return (+ n 1)))
  ;; If fast pointer eventually equals slow pointer,
  ;;  then we must be stuck in a circular list.
  ;; (A deeper property is the converse: if we are
  ;;  stuck in a circular list, then eventually the
  ;;  fast pointer will equal the slow pointer.
  ;;  That fact justifies this implementation.)
  (when (and (eq fast slow) (> n 0)) (return nil))))
```

listp *Function*

Syntax:

listp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

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

Examples:

```
(listp nil) → true
(listp (cons 1 2)) → true
(listp (make-array 6)) → false
(listp t) → false
```

See Also:

consp

Notes:

If *object* is a *cons*, **listp** does not check whether *object* is a *proper list*; it returns *true* for any kind of *list*.

```
(listp object) ≡ (typep object 'list) ≡ (typep object '(or cons null))
```

make-list *Function*

Syntax:

make-list *size* `&key` *initial-element* → *list*

Arguments and Values:

size—a non-negative *integer*.

initial-element—an *object*. The default is **nil**.

list—a *list*.

Description:

Returns a *list* of *length* given by *size*, each of the *elements* of which is *initial-element*.

Examples:

```
(make-list 5) → (NIL NIL NIL NIL NIL)
(make-list 3 :initial-element 'rah) → (RAH RAH RAH)
(make-list 2 :initial-element '(1 2 3)) → ((1 2 3) (1 2 3))
(make-list 0) → NIL ;i.e.,  ()
(make-list 0 :initial-element 'new-element) → NIL
```

Exceptional Situations:

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

See Also:

cons, **list**

push *Macro*

Syntax:

push *item place* → *new-place-value*

Arguments and Values:

item—an *object*.

place—a *place*, the *value* of which may be any *object*.

new-place-value—a *list* (the new *value* of *place*).

Description:

push prepends *item* to the *list* that is stored in *place*, stores the resulting *list* in *place*, and returns the *list*.

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

Examples:

```
(setq llst '(nil)) → (NIL)
(push 1 (car llst)) → (1)
llst → ((1))
(push 1 (car llst)) → (1 1)
llst → ((1 1))
(setq x '(a (b c) d)) → (A (B C) D)
(push 5 (cadr x)) → (5 B C)
x → (A (5 B C) D)
```

Side Effects:

The contents of *place* are modified.

See Also:

pop, **pushnew**, Section 5.1 (Generalized Reference)

Notes:

The effect of `(push` *item place*`)` is equivalent to

`(setf place (cons` *item place*`))`

except that the *subforms* of *place* are evaluated only once, and *item* is evaluated before *place*.

pop *Macro*

Syntax:

pop *place* → *element*

Arguments and Values:

place—a *place*, the *value* of which is a *list* (possibly, but necessarily, a *dotted list* or *circular list*).

element—an *object* (the *car* of the contents of *place*).

Description:

pop *reads* the *value* of *place*, remembers the *car* of the *list* which was retrieved, *writes* the *cdr* of the *list* back into the *place*, and finally *yields* the *car* of the originally retrieved *list*.

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

Examples:

```
(setq stack '(a b c)) → (A B C)
(pop stack) → A
stack → (B C)
(setq llst '((1 2 3 4))) → ((1 2 3 4))
(pop (car llst)) → 1
llst → ((2 3 4))
```

Side Effects:

The contents of *place* are modified.

See Also:

push, **pushnew**, Section 5.1 (Generalized Reference)

Notes:

The effect of (pop *place*) is roughly equivalent to

```
(prog1 (car place) (setf place (cdr place)))
```

except that the latter would evaluate any *subforms* of *place* three times, while **pop** evaluates them only once.

first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth *Accessor*

Syntax:

first *list*	→ *object*	(**setf** (**first** *list*) *new-object*)
second *list*	→ *object*	(**setf** (**second** *list*) *new-object*)
third *list*	→ *object*	(**setf** (**third** *list*) *new-object*)
fourth *list*	→ *object*	(**setf** (**fourth** *list*) *new-object*)
fifth *list*	→ *object*	(**setf** (**fifth** *list*) *new-object*)
sixth *list*	→ *object*	(**setf** (**sixth** *list*) *new-object*)
seventh *list*	→ *object*	(**setf** (**seventh** *list*) *new-object*)
eighth *list*	→ *object*	(**setf** (**eighth** *list*) *new-object*)
ninth *list*	→ *object*	(**setf** (**ninth** *list*) *new-object*)
tenth *list*	→ *object*	(**setf** (**tenth** *list*) *new-object*)

Arguments and Values:

list—a *list*, which might be a *dotted list* or a *circular list*.

first, second, third, fourth, fifth, sixth, seventh, ...

object, *new-object*—an *object*.

Description:

The functions **first**, **second**, **third**, **fourth**, **fifth**, **sixth**, **seventh**, **eighth**, **ninth**, and **tenth** *access* the first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, and tenth *elements* of *list*, respectively. Specifically,

```
(first list)    ≡  (car list)
(second list)   ≡  (car (cdr list))
(third list)    ≡  (car (cddr list))
(fourth list)   ≡  (car (cdddr list))
(fifth list)    ≡  (car (cddddr list))
(sixth list)    ≡  (car (cdr (cddddr list)))
(seventh list)  ≡  (car (cddr (cddddr list)))
(eighth list)   ≡  (car (cdddr (cddddr list)))
(ninth list)    ≡  (car (cddddr (cddddr list)))
(tenth list)    ≡  (car (cdr (cddddr (cddddr list))))
```

setf can also be used with any of these functions to change an existing component. The same equivalences apply. For example:

```
(setf (fifth list) new-object) ≡ (setf (car (cddddr list)) new-object)
```

Examples:

```
 (setq lst '(1 2 3 (4 5 6) ((V)) vi 7 8 9 10))
→ (1 2 3 (4 5 6) ((V)) VI 7 8 9 10)
 (first lst) → 1
 (tenth lst) → 10
 (fifth lst) → ((V))
 (second (fourth lst)) → 5
 (sixth '(1 2 3)) → NIL
 (setf (fourth lst) "four") → "four"
 lst → (1 2 3 "four" ((V)) VI 7 8 9 10)
```

See Also:

car, **nth**

Notes:

first is functionally equivalent to **car**, **second** is functionally equivalent to **cadr**, **third** is functionally equivalent to **caddr**, and **fourth** is functionally equivalent to **cadddr**.

The ordinal numbering used here is one-origin, as opposed to the zero-origin numbering used by **nth**:

```
(fifth x) ≡ (nth 4 x)
```

nth *Accessor*

Syntax:

nth *n list* → *object*

(**setf** (**nth** *n list*) *new-object*)

Arguments and Values:

n—a non-negative *integer*.

list—a *list*, which might be a *dotted list* or a *circular list*.

object—an *object*.

new-object—an *object*.

Description:

nth locates the *n*th element of *list*, where the *car* of the *list* is the "zeroth" element. Specifically,

(nth *n list*) ≡ (car (nthcdr *n list*))

nth may be used to specify a *place* to **setf**. Specifically,

(setf (nth *n list*) *new-object*) ≡ (setf (car (nthcdr *n list*)) *new-object*)

Examples:

```
(nth 0 '(foo bar baz)) → FOO
(nth 1 '(foo bar baz)) → BAR
(nth 3 '(foo bar baz)) → NIL
(setq 0-to-3 (list 0 1 2 3)) → (0 1 2 3)
(setf (nth 2 0-to-3) "two") → "two"
0-to-3 → (0 1 "two" 3)
```

See Also:

elt, **first**, **nthcdr**

endp *Function*

Syntax:

endp *list* → *generalized-boolean*

Arguments and Values:

list—a *list*, which might be a *dotted list* or a *circular list*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *list* is the *empty list*. Returns *false* if *list* is a *cons*.

Examples:

```
(endp nil) → true
(endp '(1 2)) → false
(endp (cddr '(1 2))) → true
```

Exceptional Situations:

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

Notes:

The purpose of **endp** is to test for the end of *proper list*. Since **endp** does not descend into a *cons*, it is well-defined to pass it a *dotted list*. However, if shorter "lists" are iteratively produced by calling **cdr** on such a *dotted list* and those "lists" are tested with **endp**, a situation that has undefined consequences will eventually result when the *non-nil atom* (which is not in fact a *list*) finally becomes the argument to **endp**. Since this is the usual way in which **endp** is used, it is conservative programming style and consistent with the intent of **endp** to treat **endp** as simply a function on *proper lists* which happens not to enforce an argument type of *proper list* except when the argument is *atomic*.

null *Function*

Syntax:

null *object* → *boolean*

Arguments and Values:

object—an *object*.

boolean—a *boolean*.

Description:

Returns **t** if *object* is the *empty list*; otherwise, returns **nil**.

Examples:

```
(null '()) → T
(null nil) → T
(null t) → NIL
(null 1) → NIL
```

See Also:

not

Notes:

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

```
(null object) ≡ (typep object 'null) ≡ (eq object '())
```

nconc *Function*

Syntax:

nconc &rest *lists* → *concatenated-list*

Arguments and Values:

list—each but the last must be a *list* (which might be a *dotted list* but must not be a *circular list*); the last *list* may be any *object*.

concatenated-list—a *list*.

Description:

Returns a *list* that is the concatenation of *lists*. If no *lists* are supplied, (nconc) returns **nil**. **nconc** is defined using the following recursive relationship:

```
(nconc) → ()
(nconc nil . lists) ≡ (nconc . lists)
(nconc list) → list
(nconc list-1 list-2) ≡ (progn (rplacd (last list-1) list-2) list-1)
(nconc list-1 list-2 . lists) ≡ (nconc (nconc list-1 list-2) . lists)
```

Examples:

```
(nconc) → NIL
(setq x '(a b c)) → (A B C)
(setq y '(d e f)) → (D E F)
(nconc x y) → (A B C D E F)
x → (A B C D E F)
```

Note, in the example, that the value of `x` is now different, since its last *cons* has been **rplacd**'d to the value of `y`. If `(nconc x y)` were evaluated again, it would yield a piece of a *circular list*, whose printed representation would be `(A B C D E F D E F D E F ...)`, repeating forever; if the ***print-circle*** switch were *non-nil*, it would be printed as `(A B C . #1=(D E F . #1#))`.

```
(setq foo (list 'a 'b 'c 'd 'e)
      bar (list 'f 'g 'h 'i 'j)
      baz (list 'k 'l 'm)) → (K L M)
(setq foo (nconc foo bar baz)) → (A B C D E F G H I J K L M)
foo → (A B C D E F G H I J K L M)
bar → (F G H I J K L M)
baz → (K L M)

(setq foo (list 'a 'b 'c 'd 'e)
      bar (list 'f 'g 'h 'i 'j)
      baz (list 'k 'l 'm)) → (K L M)
(setq foo (nconc nil foo bar nil baz)) → (A B C D E F G H I J K L M)
foo → (A B C D E F G H I J K L M)
bar → (F G H I J K L M)
baz → (K L M)
```

Side Effects:

The *lists* are modified rather than copied.

See Also:

append, **concatenate**

append *Function*

Syntax:

append `&rest` *lists* → *result*

Arguments and Values:

list—each must be a *proper list* except the last, which may be any *object*.

result—an *object*. This will be a *list* unless the last *list* was not a *list* and all preceding *lists* were *null*.

Description:

append returns a new *list* that is the concatenation of the copies. *lists* are left unchanged; the *list structure* of each of *lists* except the last is copied. The last argument is not copied; it becomes the *cdr* of the final *dotted pair* of the concatenation of the preceding *lists*, or is returned directly if there are no preceding *non-empty lists*.

Examples:

```
(append '(a b c) '(d e f) '() '(g)) → (A B C D E F G)
(append '(a b c) 'd) → (A B C . D)
(setq lst '(a b c)) → (A B C)
(append lst '(d)) → (A B C D)
lst → (A B C)
(append) → NIL
(append 'a) → A
```

See Also:

nconc, **concatenate**

revappend, nreconc *Function*

Syntax:

revappend *list tail* → *result-list*

nreconc *list tail* → *result-list*

Arguments and Values:

list—a *proper list*.

tail—an *object*.

result-list—an *object*.

Description:

revappend constructs a *copy*$_2$ of *list*, but with the *elements* in reverse order. It then appends (as if by **nconc**) the *tail* to that reversed list and returns the result.

nreconc reverses the order of *elements* in *list* (as if by **nreverse**). It then appends (as if by **nconc**) the *tail* to that reversed list and returns the result.

The resulting *list* shares *list structure* with *tail*.

revappend, nreconc

Examples:

```
(let ((list-1 (list 1 2 3))
      (list-2 (list 'a 'b 'c)))
  (print (revappend list-1 list-2))
  (print (equal list-1 '(1 2 3)))
  (print (equal list-2 '(a b c))))
▷ (3 2 1 A B C)
▷ T
▷ T
→ T

(revappend '(1 2 3) '()) → (3 2 1)
(revappend '(1 2 3) '(a . b)) → (3 2 1 A . B)
(revappend '() '(a b c)) → (A B C)
(revappend '(1 2 3) 'a) → (3 2 1 . A)
(revappend '() 'a) → A   ;degenerate case

(let ((list-1 '(1 2 3))
      (list-2 '(a b c)))
  (print (nreconc list-1 list-2))
  (print (equal list-1 '(1 2 3)))
  (print (equal list-2 '(a b c))))
▷ (3 2 1 A B C)
▷ NIL
▷ T
→ T
```

Side Effects:

revappend does not modify either of its *arguments*. **nreconc** is permitted to modify *list* but not *tail*.

Although it might be implemented differently, **nreconc** is constrained to have side-effect behavior equivalent to:

```
(nconc (nreverse list) tail)
```

See Also:

reverse, **nreverse**, **nconc**

Notes:

The following functional equivalences are true, although good *implementations* will typically use a faster algorithm for achieving the same effect:

```
(revappend list tail) ≡ (nconc (reverse list) tail)
(nreconc list tail) ≡ (nconc (nreverse list) tail)
```

butlast, nbutlast *Function*

Syntax:

butlast *list* &optional *n* → *result-list*

nbutlast *list* &optional *n* → *result-list*

Arguments and Values:

list—a *list*, which might be a *dotted list* but must not be a *circular list*.

n—a non-negative *integer*.

result-list—a *list*.

Description:

butlast returns a copy of *list* from which the last *n* conses have been omitted. If *n* is not supplied, its value is 1. If there are fewer than *n* conses in *list*, **nil** is returned and, in the case of **nbutlast**, *list* is not modified.

nbutlast is like **butlast**, but **nbutlast** may modify *list*. It changes the *cdr* of the *cons* *n*+1 from the end of the *list* to **nil**.

Examples:

```
(setq lst '(1 2 3 4 5 6 7 8 9)) → (1 2 3 4 5 6 7 8 9)
(butlast lst) → (1 2 3 4 5 6 7 8)
(butlast lst 5) → (1 2 3 4)
(butlast lst (+ 5 5)) → NIL
lst → (1 2 3 4 5 6 7 8 9)
(nbutlast lst 3) → (1 2 3 4 5 6)
lst → (1 2 3 4 5 6)
(nbutlast lst 99) → NIL
lst → (1 2 3 4 5 6)
(butlast '(a b c d)) → (A B C)
(butlast '((a b) (c d))) → ((A B))
(butlast '(a)) → NIL
(butlast nil) → NIL
(setq foo (list 'a 'b 'c 'd)) → (A B C D)
(nbutlast foo) → (A B C)
foo → (A B C)
```

```
(nbutlast (list 'a)) → NIL
(nbutlast '()) → NIL
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *list* is not a *proper list* or a *dotted list*. Should signal an error of *type* **type-error** if *n* is not a non-negative *integer*.

Notes:

(butlast *list n*) ≡ (ldiff *list* (last *list n*))

last *Function*

Syntax:

last *list* &optional *n* → *tail*

Arguments and Values:

list—a *list*, which might be a *dotted list* but must not be a *circular list*.

n—a non-negative *integer*. The default is 1.

tail—an *object*.

Description:

last returns the last *n conses* (not the last *n* elements) of *list*). If *list* is (), **last** returns ().

If *n* is zero, the atom that terminates *list* is returned. If *n* is greater than or equal to the number of *cons* cells in *list*, the result is *list*.

Examples:

```
(last nil) → NIL
(last '(1 2 3)) → (3)
(last '(1 2 . 3)) → (2 . 3)
(setq x (list 'a 'b 'c 'd)) → (A B C D)
(last x) → (D)
(rplacd (last x) (list 'e 'f)) x → (A B C D E F)
(last x) → (F)

(last '(a b c))   → (C)

(last '(a b c) 0) → ()
(last '(a b c) 1) → (C)
```

```
(last '(a b c) 2) → (B C)
(last '(a b c) 3) → (A B C)
(last '(a b c) 4) → (A B C)

(last '(a . b) 0) → B
(last '(a . b) 1) → (A . B)
(last '(a . b) 2) → (A . B)
```

Exceptional Situations:

The consequences are undefined if *list* is a *circular list*. Should signal an error of *type* **type-error** if *n* is not a non-negative *integer*.

See Also:

butlast, **nth**

Notes:

The following code could be used to define **last**.

```
(defun last (list &optional (n 1))
  (check-type n (integer 0))
  (do ((l list (cdr l))
       (r list)
       (i 0 (+ i 1)))
      ((atom l) r)
    (if (>= i n) (pop r))))
```

ldiff, tailp *Function*

Syntax:

ldiff *list object* → *result-list*

tailp *object list* → *generalized-boolean*

Arguments and Values:

list—a *list*, which might be a *dotted list*.

object—an *object*.

result-list—a *list*.

generalized-boolean—a *generalized boolean*.

ldiff, tailp

Description:

If *object* is the *same* as some *tail* of *list*, **tailp** returns *true*; otherwise, it returns *false*.

If *object* is the *same* as some *tail* of *list*, **ldiff** returns a *fresh list* of the *elements* of *list* that precede **object** in the *list structure* of *list*; otherwise, it returns a $copy_2$ of *list*.

Examples:

```
(let ((lists '#((a b c) (a b c . d))))
  (dotimes (i (length lists)) ()
    (let ((list (aref lists i)))
      (format t "~2&list=~S ~21T(tailp object list)~
                 ~44T(ldiff list object)~%" list)
        (let ((objects (vector list (cddr list) (copy-list (cddr list))
                               '(f g h) '() 'd 'x)))
          (dotimes (j (length objects)) ()
            (let ((object (aref objects j)))
              (format t "~& object=~S ~21T~S ~44T~S"
                      object (tailp object list) (ldiff list object))))))))
▷
▷ list=(A B C)         (tailp object list)    (ldiff list object)
▷  object=(A B C)      T                      NIL
▷  object=(C)          T                      (A B)
▷  object=(C)          NIL                    (A B C)
▷  object=(F G H)      NIL                    (A B C)
▷  object=NIL          T                      (A B C)
▷  object=D            NIL                    (A B C)
▷  object=X            NIL                    (A B C)
▷
▷ list=(A B C . D)     (tailp object list)    (ldiff list object)
▷  object=(A B C . D)  T                      NIL
▷  object=(C . D)      T                      (A B)
▷  object=(C . D)      NIL                    (A B C . D)
▷  object=(F G H)      NIL                    (A B C . D)
▷  object=NIL          NIL                    (A B C . D)
▷  object=D            T                      (A B C)
▷  object=X            NIL                    (A B C . D)
→ NIL
```

Side Effects:

Neither **ldiff** nor **tailp** modifies either of its *arguments*.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list* is not a *proper list* or a *dotted list*.

See Also:

set-difference

Notes:

If the *list* is a *circular list*, **tailp** will reliably *yield* a *value* only if the given *object* is in fact a *tail* of *list*. Otherwise, the consequences are unspecified: a given *implementation* which detects the circularity must return *false*, but since an *implementation* is not obliged to detect such a *situation*, **tailp** might just loop indefinitely without returning in that case.

tailp could be defined as follows:

```
(defun tailp (object list)
  (do ((list list (cdr list)))
      ((atom list) (eql list object))
     (if (eql object list)
         (return t))))
```

and **ldiff** could be defined by:

```
(defun ldiff (list object)
  (do ((list list (cdr list))
       (r '() (cons (car list) r)))
      ((atom list)
       (if (eql list object) (nreverse r) (nreconc r list)))
    (when (eql object list)
      (return (nreverse r)))))
```

nthcdr *Function*

Syntax:

nthcdr *n list* → *tail*

Arguments and Values:

n—a non-negative *integer*.

list—a *list*, which might be a *dotted list* or a *circular list*.

tail—an *object*.

Description:

Returns the *tail* of *list* that would be obtained by calling **cdr** *n* times in succession.

Examples:

```
(nthcdr 0 '()) → NIL
(nthcdr 3 '()) → NIL
(nthcdr 0 '(a b c)) → (A B C)
(nthcdr 2 '(a b c)) → (C)
(nthcdr 4 '(a b c)) → ()
(nthcdr 1 '(0 . 1)) → 1

(locally (declare (optimize (safety 3)))
  (nthcdr 3 '(0 . 1)))
Error: Attempted to take CDR of 1.
```

Exceptional Situations:

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

For *n* being an integer greater than `1`, the error checking done by `(nthcdr` *n list*`)` is the same as for `(nthcdr (-` *n* `1) (cdr` *list*`))`; see the *function* **cdr**.

See Also:

cdr, **nth**, **rest**

rest *Accessor*

Syntax:

rest *list* → *tail*

(setf (rest *list***)** *new-tail***)**

Arguments and Values:

list—a *list*, which might be a *dotted list* or a *circular list*.

tail—an *object*.

Description:

rest performs the same operation as **cdr**, but mnemonically complements **first**. Specifically,

```
(rest list) ≡ (cdr list)
(setf (rest list) new-tail) ≡ (setf (cdr list) new-tail)
```

Examples:

```
(rest '(1 2)) → (2)
```

```
(rest '(1 . 2)) → 2
(rest '(1)) → NIL
(setq *cons* '(1 . 2)) → (1 . 2)
(setf (rest *cons*) "two") → "two"
*cons* → (1 . "two")
```

See Also:

cdr, **nthcdr**

Notes:

rest is often preferred stylistically over **cdr** when the argument is to being subjectively viewed as a *list* rather than as a *cons*.

member, member-if, member-if-not *Function*

Syntax:

member *item list* &key *key test test-not* → *tail*

member-if *predicate list* &key *key* → *tail*

member-if-not *predicate list* &key *key* → *tail*

Arguments and Values:

item—an *object*.

list—a *proper list*.

predicate—a *designator* for a *function* of one *argument* that returns a *generalized boolean*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

tail—a *list*.

Description:

member, **member-if**, and **member-if-not** each search *list* for *item* or for a top-level element that *satisfies the test*. The argument to the *predicate* function is an element of *list*.

If some element *satisfies the test*, the tail of *list* beginning with this element is returned; otherwise **nil** is returned.

list is searched on the top level only.

Examples:

```
(member 2 '(1 2 3)) → (2 3)
(member 2 '((1 . 2) (3 . 4)) :test-not #'= :key #'cdr) → ((3 . 4))
(member 'e '(a b c d)) → NIL

(member-if #'listp '(a b nil c d)) → (NIL C D)
(member-if #'numberp '(a #\Space 5/3 foo)) → (5/3 FOO)
(member-if-not #'zerop
               '(3 6 9 11 . 12)
               :key #'(lambda (x) (mod x 3))) → (11 . 12)
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list* is not a *proper list*.

See Also:

find, **position**, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

The *function* **member-if-not** is deprecated.

In the following

```
(member 'a '(g (a y) c a d e a f)) → (A D E A F)
```

the value returned by **member** is *identical* to the portion of the *list* beginning with `a`. Thus **rplaca** on the result of **member** can be used to alter the part of the *list* where `a` was found (assuming a check has been made that **member** did not return **nil**).

mapc, mapcar, mapcan, mapl, maplist, mapcon

Function

Syntax:

mapc *function* `&rest` *lists*$^+$ → *list-1*

mapcar *function* `&rest` *lists*$^+$ → *result-list*

mapcan *function* `&rest` *lists*$^+$ → *concatenated-results*

mapl *function* `&rest` *lists*$^+$ → *list-1*

maplist *function* `&rest` *lists*$^+$ → *result-list*

mapc, mapcar, mapcan, mapl, maplist, mapcon

mapcon *function* &rest *lists*$^+$ → *concatenated-results*

Arguments and Values:

function—a *designator* for a *function* that must take as many *arguments* as there are *lists*.

list—a *proper list*.

list-1—the first *list* (which must be a *proper list*).

result-list—a *list*.

concatenated-results—a *list*.

Description:

The mapping operation involves applying *function* to successive sets of arguments in which one argument is obtained from each *sequence*. Except for **mapc** and **mapl**, the result contains the results returned by *function*. In the cases of **mapc** and **mapl**, the resulting *sequence* is *list*.

function is called first on all the elements with index 0, then on all those with index 1, and so on. *result-type* specifies the *type* of the resulting *sequence*. If *function* is a *symbol*, it is **coerce**d to a *function* as if by **symbol-function**.

mapcar operates on successive *elements* of the *lists*. *function* is applied to the first *element* of each *list*, then to the second *element* of each *list*, and so on. The iteration terminates when the shortest *list* runs out, and excess elements in other lists are ignored. The value returned by **mapcar** is a *list* of the results of successive calls to *function*.

mapc is like **mapcar** except that the results of applying *function* are not accumulated. The *list* argument is returned.

maplist is like **mapcar** except that *function* is applied to successive sublists of the *lists*. *function* is first applied to the *lists* themselves, and then to the *cdr* of each *list*, and then to the *cdr* of the *cdr* of each *list*, and so on.

mapl is like **maplist** except that the results of applying *function* are not accumulated; *list-1* is returned.

mapcan and **mapcon** are like **mapcar** and **maplist** respectively, except that the results of applying *function* are combined into a *list* by the use of **nconc** rather than **list**. That is,

```
(mapcon f x1 ... xn)
  ≡ (apply #'nconc (maplist f x1 ... xn))
```

and similarly for the relationship between **mapcan** and **mapcar**.

Examples:

```
(mapcar #'car '((1 a) (2 b) (3 c))) → (1 2 3)
(mapcar #'abs '(3 -4 2 -5 -6)) → (3 4 2 5 6)
```

```
(mapcar #'cons '(a b c) '(1 2 3)) → ((A . 1) (B . 2) (C . 3))

 (maplist #'append '(1 2 3 4) '(1 2) '(1 2 3))
→ ((1 2 3 4 1 2 1 2 3) (2 3 4 2 2 3))
 (maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
→ ((FOO A B C D) (FOO B C D) (FOO C D) (FOO D))
 (maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1)) '(a b a c d b c))
→ (0 0 1 0 1 1 1)
;An entry is 1 if the corresponding element of the input
;  list was the last instance of that element in the input list.

 (setq dummy nil) → NIL
 (mapc #'(lambda (&rest x) (setq dummy (append dummy x)))
        '(1 2 3 4)
        '(a b c d e)
        '(x y z)) → (1 2 3 4)
 dummy → (1 A X 2 B Y 3 C Z)

 (setq dummy nil) → NIL
 (mapl #'(lambda (x) (push x dummy)) '(1 2 3 4)) → (1 2 3 4)
 dummy → ((4) (3 4) (2 3 4) (1 2 3 4))

 (mapcan #'(lambda (x y) (if (null x) nil (list x y)))
          '(nil nil nil d e)
          '(1 2 3 4 5 6)) → (D 4 E 5)
 (mapcan #'(lambda (x) (and (numberp x) (list x)))
          '(a 1 b c 3 4 d 5))
→ (1 3 4 5)
```

In this case the function serves as a filter; this is a standard Lisp idiom using **mapcan**.

```
(mapcon #'list '(1 2 3 4)) → ((1 2 3 4) (2 3 4) (3 4) (4))
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if any *list* is not a *proper list*.

See Also:

dolist, **map**, Section 3.6 (Traversal Rules and Side Effects)

acons *Function*

Syntax:

acons *key datum alist* → *new-alist*

Arguments and Values:

key—an *object*.

datum—an *object*.

alist—an *association list*.

new-alist—an *association list*.

Description:

Creates a *fresh cons*, the *cdr* of which is *alist* and the *car* of which is another *fresh cons*, the *car* of which is *key* and the *cdr* of which is *datum*.

Examples:

```
(setq alist '()) → NIL
(acons 1 "one" alist) → ((1 . "one"))
alist → NIL
(setq alist (acons 1 "one" (acons 2 "two" alist))) → ((1 . "one") (2 . "two"))
(assoc 1 alist) → (1 . "one")
(setq alist (acons 1 "uno" alist)) → ((1 . "uno") (1 . "one") (2 . "two"))
(assoc 1 alist) → (1 . "uno")
```

See Also:

assoc, **pairlis**

Notes:

(acons *key datum alist*) ≡ (cons (cons *key datum*) *alist*)

assoc, assoc-if, assoc-if-not *Function*

Syntax:

assoc *item alist* &key *key test test-not* → *entry*

assoc-if *predicate alist* &key *key* → *entry*

assoc, assoc-if, assoc-if-not

assoc-if-not *predicate alist* &key *key* → *entry*

Arguments and Values:

item—an *object*.

alist—an *association list*.

predicate—a *designator* for a *function* of one *argument* that returns a *generalized boolean*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

entry—a *cons* that is an *element* of *alist*, or **nil**.

Description:

assoc, **assoc-if**, and **assoc-if-not** return the first *cons* in *alist* whose *car satisfies the test*, or **nil** if no such *cons* is found.

For **assoc**, **assoc-if**, and **assoc-if-not**, if **nil** appears in *alist* in place of a pair, it is ignored.

Examples:

```
 (setq values '((x . 100) (y . 200) (z . 50))) → ((X . 100) (Y . 200) (Z . 50))
 (assoc 'y values) → (Y . 200)
 (rplacd (assoc 'y values) 201) → (Y . 201)
 (assoc 'y values) → (Y . 201)
 (setq alist '((1 . "one")(2 . "two")(3 . "three")))
→ ((1 . "one") (2 . "two") (3 . "three"))
 (assoc 2 alist) → (2 . "two")
 (assoc-if #'evenp alist) → (2 . "two")
 (assoc-if-not #'(lambda(x) (< x 3)) alist) → (3 . "three")
 (setq alist '(("one" . 1)("two" . 2))) → (("one" . 1) ("two" . 2))
 (assoc "one" alist) → NIL
 (assoc "one" alist :test #'equalp) → ("one" . 1)
 (assoc "two" alist :key #'(lambda(x) (char x 2))) → NIL
 (assoc #\o alist :key #'(lambda(x) (char x 2))) → ("two" . 2)
 (assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z))) →  (R . X)
 (assoc 'goo '((foo . bar) (zoo . goo))) → NIL
 (assoc '2 '((1 a b c) (2 b c d) (-7 x y z))) → (2 B C D)
 (setq alist '(("one" . 1) ("2" . 2) ("three" . 3)))
→ (("one" . 1) ("2" . 2) ("three" . 3))
 (assoc-if-not #'alpha-char-p alist
              :key #'(lambda (x) (char x 0))) → ("2" . 2)
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *alist* is not an *association list*.

See Also:

rassoc, **find**, **member**, **position**, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

The *function* **assoc-if-not** is deprecated.

It is possible to **rplacd** the result of **assoc**, provided that it is not **nil**, in order to "update" *alist*.

The two expressions

```
(assoc item list :test fn)
```

and

```
(find item list :test fn :key #'car)
```

are equivalent in meaning with one exception: if **nil** appears in *alist* in place of a pair, and *item* is **nil**, **find** will compute the *car* of the **nil** in *alist*, find that it is equal to *item*, and return **nil**, whereas **assoc** will ignore the **nil** in *alist* and continue to search for an actual *cons* whose *car* is **nil**.

copy-alist *Function*

Syntax:

copy-alist *alist* → *new-alist*

Arguments and Values:

alist—an *association list*.

new-alist—an *association list*.

Description:

copy-alist returns a *copy* of *alist*.

The *list structure* of *alist* is copied, and the *elements* of *alist* which are *conses* are also copied (as *conses* only). Any other *objects* which are referred to, whether directly or indirectly, by the *alist* continue to be shared.

Examples:

```
(defparameter *alist* (acons 1 "one" (acons 2 "two" '())))
*alist* → ((1 . "one") (2 . "two"))
(defparameter *list-copy* (copy-list *alist*))
*list-copy* → ((1 . "one") (2 . "two"))
(defparameter *alist-copy* (copy-alist *alist*))
*alist-copy* → ((1 . "one") (2 . "two"))
(setf (cdr (assoc 2 *alist-copy*)) "deux") → "deux"
*alist-copy* → ((1 . "one") (2 . "deux"))
*alist* → ((1 . "one") (2 . "two"))
(setf (cdr (assoc 1 *list-copy*)) "uno") → "uno"
*list-copy* → ((1 . "uno") (2 . "two"))
*alist* → ((1 . "uno") (2 . "two"))
```

See Also:

copy-list

pairlis *Function*

Syntax:

pairlis *keys data* &optional *alist* → *new-alist*

Arguments and Values:

keys—a *proper list*.

data—a *proper list*.

alist—an *association list*. The default is the *empty list*.

new-alist—an *association list*.

Description:

Returns an *association list* that associates elements of *keys* to corresponding elements of *data*. The consequences are undefined if *keys* and *data* are not of the same *length*.

If *alist* is supplied, **pairlis** returns a modified *alist* with the new pairs prepended to it. The new pairs may appear in the resulting *association list* in either forward or backward order. The result of

```
(pairlis '(one two) '(1 2) '((three . 3) (four . 19)))
```

might be

```
((one . 1) (two . 2) (three . 3) (four . 19))
```

or

```
((two . 2) (one . 1) (three . 3) (four . 19))
```

Examples:

```
(setq keys '(1 2 3)
       data '("one" "two" "three")
       alist '((4 . "four"))) → ((4 . "four"))
(pairlis keys data) → ((3 . "three") (2 . "two") (1 . "one"))
(pairlis keys data alist)
→ ((3 . "three") (2 . "two") (1 . "one") (4 . "four"))
alist → ((4 . "four"))
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *keys* and *data* are not *proper lists*.

See Also:

acons

rassoc, rassoc-if, rassoc-if-not *Function*

Syntax:

rassoc *item alist* &key *key test test-not* → *entry*

rassoc-if *predicate alist* &key *key* → *entry*

rassoc-if-not *predicate alist* &key *key* → *entry*

Arguments and Values:

item—an *object*.

alist—an *association list*.

predicate—a *designator* for a *function* of one *argument* that returns a *generalized boolean*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

entry—a *cons* that is an *element* of the *alist*, or **nil**.

Description:

rassoc, **rassoc-if**, and **rassoc-if-not** return the first *cons* whose *cdr satisfies the test*. If no such *cons* is found, **nil** is returned.

If **nil** appears in *alist* in place of a pair, it is ignored.

Examples:

```
 (setq alist '((1 . "one") (2 . "two") (3 . 3)))
→ ((1 . "one") (2 . "two") (3 . 3))
 (rassoc 3 alist) → (3 . 3)
 (rassoc "two" alist) → NIL
 (rassoc "two" alist :test 'equal) → (2 . "two")
 (rassoc 1 alist :key #'(lambda (x) (if (numberp x) (/ x 3)))) → (3 . 3)
 (rassoc 'a '((a . b) (b . c) (c . a) (z . a))) → (C . A)
 (rassoc-if #'stringp alist) → (1 . "one")
 (rassoc-if-not #'vectorp alist) → (3 . 3)
```

See Also:

assoc, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

The *function* **rassoc-if-not** is deprecated.

It is possible to **rplaca** the result of **rassoc**, provided that it is not **nil**, in order to "update" *alist*.

The expressions

```
 (rassoc item list :test fn)
```

and

```
 (find item list :test fn :key #'cdr)
```

are equivalent in meaning, except when the `item` is **nil** and **nil** appears in place of a pair in the *alist*. See the *function* **assoc**.

get-properties *Function*

Syntax:

get-properties *plist indicator-list* → *indicator, value, tail*

Arguments and Values:

plist—a *property list*.

indicator-list—a *proper list* (of *indicators*).

indicator—an *object* that is an *element* of *indicator-list*.

value—an *object*.

tail—a *list*.

Description:

get-properties is used to look up any of several *property list* entries all at once.

It searches the *plist* for the first entry whose *indicator* is *identical* to one of the *objects* in *indicator-list*. If such an entry is found, the *indicator* and *value* returned are the *property indicator* and its associated *property value*, and the *tail* returned is the *tail* of the *plist* that begins with the found entry (*i.e.*, whose *car* is the *indicator*). If no such entry is found, the *indicator*, *value*, and *tail* are all **nil**.

Examples:

```
(setq x '()) → NIL
(setq *indicator-list* '(prop1 prop2)) → (PROP1 PROP2)
(getf x 'prop1) → NIL
(setf (getf x 'prop1) 'val1) → VAL1
(eq (getf x 'prop1) 'val1) → true
(get-properties x *indicator-list*) → PROP1, VAL1, (PROP1 VAL1)
x → (PROP1 VAL1)
```

See Also:

get, **getf**

getf *Accessor*

Syntax:

getf *plist indicator* &optional *default* → *value*

(**setf** (**getf** *place indicator* &optional *default*) *new-value*)

Arguments and Values:

plist—a *property list*.

place—a *place*, the *value* of which is a *property list*.

getf

indicator—an *object*.

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

value—an *object*.

new-value—an *object*.

Description:

getf finds a *property* on the *plist* whose *property indicator* is *identical* to *indicator*, and returns its corresponding *property value*. If there are multiple *properties*$_1$ with that *property indicator*, **getf** uses the first such *property*. If there is no *property* with that *property indicator*, *default* is returned.

setf of **getf** may be used to associate a new *object* with an existing indicator in the *property list* held by *place*, or to create a new assocation if none exists. If there are multiple *properties*$_1$ with that *property indicator*, **setf** of **getf** associates the *new-value* with the first such *property*. When a **getf** *form* is used as a **setf** *place*, any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its *value* is ignored.

setf of **getf** is permitted to either *write* the *value* of *place* itself, or modify of any part, *car* or *cdr*, of the *list structure* held by *place*.

Examples:

```
 (setq x '()) → NIL
 (getf x 'prop1) → NIL
 (getf x 'prop1 7) → 7
 (getf x 'prop1) → NIL
 (setf (getf x 'prop1) 'val1) → VAL1
 (eq (getf x 'prop1) 'val1) → true
 (getf x 'prop1) → VAL1
 (getf x 'prop1 7) → VAL1
 x → (PROP1 VAL1)

;; Examples of implementation variation permitted.
 (setq foo (list 'a 'b 'c 'd 'e 'f)) → (A B C D E F)
 (setq bar (cddr foo)) → (C D E F)
 (remf foo 'c) → true
 foo → (A B E F)
 bar
→ (C D E F)
or→ (C)
or→ (NIL)
or→ (C NIL)
or→ (C D)
```

See Also:

get, **get-properties**, **setf**, Section 5.1.2.2 (Function Call Forms as Places)

Notes:

There is no way (using **getf**) to distinguish an absent property from one whose value is *default*; but see **get-properties**.

Note that while supplying a *default* argument to **getf** in a **setf** situation is sometimes not very interesting, it is still important because some macros, such as **push** and **incf**, require a *place* argument which data is both *read* from and *written* to. In such a context, if a *default* argument is to be supplied for the *read* situation, it must be syntactically valid for the *write* situation as well. For example,

```
(let ((plist '()))
  (incf (getf plist 'count 0))
  plist) → (COUNT 1)
```

remf *Macro*

Syntax:

remf *place indicator* → *generalized-boolean*

Arguments and Values:

place—a *place*.

indicator—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

remf removes from the *property list* stored in *place* a *property*$_1$ with a *property indicator identical* to *indicator*. If there are multiple *properties*$_1$ with the *identical* key, **remf** only removes the first such *property*. **remf** returns *false* if no such *property* was found, or *true* if a property was found.

The *property indicator* and the corresponding *property value* are removed in an undefined order by destructively splicing the property list. **remf** is permitted to either **setf** *place* or to **setf** any part, **car** or **cdr**, of the *list structure* held by that *place*.

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

Examples:

```
(setq x (cons () ())) → (NIL)
(setf (getf (car x) 'prop1) 'val1) → VAL1
(remf (car x) 'prop1) → true
(remf (car x) 'prop1) → false
```

Side Effects:

The property list stored in *place* is modified.

See Also:

remprop, **getf**

intersection, nintersection *Function*

Syntax:

intersection *list-1 list-2* &key *key test test-not* → *result-list*

nintersection *list-1 list-2* &key *key test test-not* → *result-list*

Arguments and Values:

list-1—a *proper list*.

list-2—a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

result-list—a *list*.

Description:

intersection and **nintersection** return a *list* that contains every element that occurs in both *list-1* and *list-2*.

nintersection is the destructive version of **intersection**. It performs the same operation, but may destroy *list-1* using its cells to construct the result. *list-2* is not destroyed.

The intersection operation is described as follows. For all possible ordered pairs consisting of one *element* from *list-1* and one *element* from *list-2*, `:test` or `:test-not` are used to determine whether they *satisfy the test*. The first argument to the `:test` or `:test-not` function is an element of *list-1*; the second argument is an element of *list-2*. If `:test` or `:test-not` is not supplied, **eql** is used. It is an error if `:test` and `:test-not` are supplied in the same function call.

intersection, nintersection

If `:key` is supplied (and not **nil**), it is used to extract the part to be tested from the *list* element. The argument to the `:key` function is an element of either *list-1* or *list-2*; the `:key` function typically returns part of the supplied element. If `:key` is not supplied or **nil**, the *list-1* and *list-2* elements are used.

For every pair that *satifies the test*, exactly one of the two elements of the pair will be put in the result. No element from either *list* appears in the result that does not *satisfy the test* for an element from the other *list*. If one of the *lists* contains duplicate elements, there may be duplication in the result.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The result *list* may share cells with, or be **eq** to, either *list-1* or *list-2* if appropriate.

Examples:

```
 (setq list1 (list 1 1 2 3 4 a b c "A" "B" "C" "d")
       list2 (list 1 4 5 b c d "a" "B" "c" "D"))
  → (1 4 5 B C D "a" "B" "c" "D")
 (intersection list1 list2) → (C B 4 1 1)
 (intersection list1 list2 :test 'equal) → ("B" C B 4 1 1)
 (intersection list1 list2 :test #'equalp) → ("d" "C" "B" "A" C B 4 1 1)
 (nintersection list1 list2) → (1 1 4 B C)
 list1 → implementation-dependent ;e.g.,  (1 1 4 B C)
 list2 → implementation-dependent ;e.g.,  (1 4 5 B C D "a" "B" "c" "D")
 (setq list1 (copy-list '((1 . 2) (2 . 3) (3 . 4) (4 . 5))))
→ ((1 . 2) (2 . 3) (3 . 4) (4 . 5))
 (setq list2 (copy-list '((1 . 3) (2 . 4) (3 . 6) (4 . 8))))
→ ((1 . 3) (2 . 4) (3 . 6) (4 . 8))
 (nintersection list1 list2 :key #'cdr) → ((2 . 3) (3 . 4))
 list1 → implementation-dependent ;e.g.,  ((1 . 2) (2 . 3) (3 . 4))
 list2 → implementation-dependent ;e.g.,  ((1 . 3) (2 . 4) (3 . 6) (4 . 8))
```

Side Effects:

nintersection can modify *list-1*, but not *list-2*.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list-1* and *list-2* are not *proper lists*.

See Also:

union, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

Since the **nintersection** side effect is not required, it should not be used in for-effect-only positions in portable code.

adjoin *Function*

Syntax:

adjoin *item list* &key *key test test-not* → *new-list*

Arguments and Values:

item—an *object*.

list—a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

new-list—a *list*.

Description:

Tests whether *item* is the same as an existing element of *list*. If the *item* is not an existing element, **adjoin** adds it to *list* (as if by **cons**) and returns the resulting *list*; otherwise, nothing is added and the original *list* is returned.

The *test*, *test-not*, and *key* affect how it is determined whether *item* is the same as an *element* of *list*. For details, see Section 17.2.1 (Satisfying a Two-Argument Test).

Examples:

```
(setq slist '()) → NIL
(adjoin 'a slist) → (A)
slist → NIL
(setq slist (adjoin '(test-item 1) slist)) → ((TEST-ITEM 1))
(adjoin '(test-item 1) slist) → ((TEST-ITEM 1) (TEST-ITEM 1))
(adjoin '(test-item 1) slist :test 'equal) → ((TEST-ITEM 1))
(adjoin '(new-test-item 1) slist :key #'cadr) → ((TEST-ITEM 1))
(adjoin '(new-test-item 1) slist) → ((NEW-TEST-ITEM 1) (TEST-ITEM 1))
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list* is not a *proper list*.

See Also:

pushnew, Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

```
(adjoin item list :key fn)
  ≡ (if (member (fn item) list :key fn) list (cons item list))
```

pushnew *Macro*

Syntax:

pushnew *item place* `&key` *key test test-not*
→ *new-place-value*

Arguments and Values:

item—an *object*.

place—a *place*, the *value* of which is a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

new-place-value—a *list* (the new *value* of *place*).

Description:

pushnew tests whether *item* is the same as any existing element of the *list* stored in *place*. If *item* is not, it is prepended to the *list*, and the new *list* is stored in *place*.

pushnew returns the new *list* that is stored in *place*.

Whether or not *item* is already a member of the *list* that is in *place* is determined by comparisons using `:test` or `:test-not`. The first argument to the `:test` or `:test-not` function is *item*; the second argument is an element of the *list* in *place* as returned by the `:key` function (if supplied).

If `:key` is supplied, it is used to extract the part to be tested from both *item* and the *list* element, as for **adjoin**.

The argument to the `:key` function is an element of the *list* stored in *place*. The `:key` function typically returns part part of the element of the *list*. If `:key` is not supplied or **nil**, the *list* element is used.

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

It is *implementation-dependent* whether or not **pushnew** actually executes the storing form for its *place* in the situation where the *item* is already a member of the *list* held by *place*.

Examples:

```
(setq x '(a (b c) d)) → (A (B C) D)
(pushnew 5 (cadr x)) → (5 B C)
x → (A (5 B C) D)
(pushnew 'b (cadr x)) → (5 B C)
x → (A (5 B C) D)
(setq lst '((1) (1 2) (1 2 3))) → ((1) (1 2) (1 2 3))
(pushnew '(2) lst) → ((2) (1) (1 2) (1 2 3))
(pushnew '(1) lst) → ((1) (2) (1) (1 2) (1 2 3))
(pushnew '(1) lst :test 'equal) → ((1) (2) (1) (1 2) (1 2 3))
(pushnew '(1) lst :key #'car) → ((1) (2) (1) (1 2) (1 2 3))
```

Side Effects:

The contents of *place* may be modified.

See Also:

push, **adjoin**, Section 5.1 (Generalized Reference)

Notes:

The effect of `(pushnew item place :test p)`

is roughly equivalent to `(setf place (adjoin item place :test p))`

except that the *subforms* of `place` are evaluated only once, and `item` is evaluated before `place`.

set-difference, nset-difference *Function*

Syntax:

set-difference *list-1 list-2* `&key` *key test test-not* → *result-list*

nset-difference *list-1 list-2* `&key` *key test test-not* → *result-list*

Arguments and Values:

list-1—a *proper list*.

list-2—a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

set-difference, nset-difference

key—a *designator* for a *function* of one argument, or **nil**.

result-list—a *list*.

Description:

set-difference returns a *list* of elements of *list-1* that do not appear in *list-2*.

nset-difference is the destructive version of **set-difference**. It may destroy *list-1*.

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, the `:test` or `:test-not` function is used to determine whether they *satisfy the test*. The first argument to the `:test` or `:test-not` function is the part of an element of *list-1* that is returned by the `:key` function (if supplied); the second argument is the part of an element of *list-2* that is returned by the `:key` function (if supplied).

If `:key` is supplied, its argument is a *list-1* or *list-2* element. The `:key` function typically returns part of the supplied element. If `:key` is not supplied, the *list-1* or *list-2* element is used.

An element of *list-1* appears in the result if and only if it does not match any element of *list-2*.

There is no guarantee that the order of elements in the result will reflect the ordering of the arguments in any particular way. The result *list* may share cells with, or be **eq** to, either of *list-1* or *list-2*, if appropriate.

Examples:

```
 (setq lst1 (list "A" "b" "C" "d")
       lst2 (list "a" "B" "C" "d")) → ("a" "B" "C" "d")
 (set-difference lst1 lst2) → ("d" "C" "b" "A")
 (set-difference lst1 lst2 :test 'equal) → ("b" "A")
 (set-difference lst1 lst2 :test #'equalp) → NIL
 (nset-difference lst1 lst2 :test #'string=) → ("A" "b")
 (setq lst1 '(("a" . "b") ("c" . "d") ("e" . "f")))
→ (("a" . "b") ("c" . "d") ("e" . "f"))
 (setq lst2 '(("c" . "a") ("e" . "b") ("d" . "a")))
→ (("c" . "a") ("e" . "b") ("d" . "a"))
 (nset-difference lst1 lst2 :test #'string= :key #'cdr)
→ (("c" . "d") ("e" . "f"))
 lst1 → (("a" . "b") ("c" . "d") ("e" . "f"))
 lst2 → (("c" . "a") ("e" . "b") ("d" . "a"))

;; Remove all flavor names that contain "c" or "w".
 (set-difference '("strawberry" "chocolate" "banana"
                   "lemon" "pistachio" "rhubarb")
          '(#\c #\w)
          :test #'(lambda (s c) (find c s)))
→ ("banana" "rhubarb" "lemon")    ;One possible ordering.
```

Side Effects:

nset-difference may destroy *list-1*.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list-1* and *list-2* are not *proper lists*.

See Also:

Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

set-exclusive-or, nset-exclusive-or *Function*

Syntax:

set-exclusive-or *list-1 list-2* `&key` *key test test-not* → *result-list*

nset-exclusive-or *list-1 list-2* `&key` *key test test-not* → *result-list*

Arguments and Values:

list-1—a *proper list*.

list-2—a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

result-list—a *list*.

Description:

set-exclusive-or returns a *list* of elements that appear in exactly one of *list-1* and *list-2*.

nset-exclusive-or is the *destructive* version of **set-exclusive-or**.

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, the `:test` or `:test-not` function is used to determine whether they *satisfy the test*.

If `:key` is supplied, it is used to extract the part to be tested from the *list-1* or *list-2* element. The first argument to the `:test` or `:test-not` function is the part of an element of *list-1* extracted by the `:key` function (if supplied); the second argument is the part of an element of *list-2* extracted by the `:key` function (if supplied). If `:key` is not supplied or **nil**, the *list-1* or *list-2* element is used.

The result contains precisely those elements of *list-1* and *list-2* that appear in no matching pair.

The result *list* of **set-exclusive-or** might share storage with one of *list-1* or *list-2*.

Examples:

```
 (setq lst1 (list 1 "a" "b")
       lst2 (list 1 "A" "b")) → (1 "A" "b")
 (set-exclusive-or lst1 lst2) → ("b" "A" "b" "a")
 (set-exclusive-or lst1 lst2 :test #'equal) → ("A" "a")
 (set-exclusive-or lst1 lst2 :test 'equalp) → NIL
 (nset-exclusive-or lst1 lst2) → ("a" "b" "A" "b")
 (setq lst1 (list (("a" . "b") ("c" . "d") ("e" . "f"))))
→ (("a" . "b") ("c" . "d") ("e" . "f"))
 (setq lst2 (list (("c" . "a") ("e" . "b") ("d" . "a"))))
→ (("c" . "a") ("e" . "b") ("d" . "a"))
 (nset-exclusive-or lst1 lst2 :test #'string= :key #'cdr)
→ (("c" . "d") ("e" . "f") ("c" . "a") ("d" . "a"))
 lst1 → (("a" . "b") ("c" . "d") ("e" . "f"))
 lst2 → (("c" . "a") ("d" . "a"))
```

Side Effects:

nset-exclusive-or is permitted to modify any part, *car* or *cdr*, of the *list structure* of *list-1* or *list-2*.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list-1* and *list-2* are not *proper lists*.

See Also:

Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

Since the **nset-exclusive-or** side effect is not required, it should not be used in for-effect-only positions in portable code.

subsetp *Function*

Syntax:

subsetp *list-1 list-2* `&key` *key test test-not* → *generalized-boolean*

Arguments and Values:

list-1—a *proper list*.

list-2—a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

generalized-boolean—a *generalized boolean*.

Description:

subsetp returns *true* if every element of *list-1* matches some element of *list-2*, and *false* otherwise.

Whether a list element is the same as another list element is determined by the functions specified by the keyword arguments. The first argument to the `:test` or `:test-not` function is typically part of an element of *list-1* extracted by the `:key` function; the second argument is typically part of an element of *list-2* extracted by the `:key` function.

The argument to the `:key` function is an element of either *list-1* or *list-2*; the return value is part of the element of the supplied list element. If `:key` is not supplied or **nil**, the *list-1* or *list-2* element itself is supplied to the `:test` or `:test-not` function.

Examples:

```
(setq cosmos '(1 "a" (1 2))) → (1 "a" (1 2))
(subsetp '(1) cosmos) → true
(subsetp '((1 2)) cosmos) → false
(subsetp '((1 2)) cosmos :test 'equal) → true
(subsetp '(1 "A") cosmos :test #'equalp) → true
(subsetp '((1) (2)) '((1) (2))) → false
(subsetp '((1) (2)) '((1) (2)) :key #'car) → true
```

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list-1* and *list-2* are not *proper lists*.

See Also:

Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

union, nunion *Function*

Syntax:

union *list-1 list-2* &key *key test test-not* → *result-list*

nunion *list-1 list-2* &key *key test test-not* → *result-list*

Arguments and Values:

list-1—a *proper list*.

list-2—a *proper list*.

test—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

test-not—a *designator* for a *function* of two *arguments* that returns a *generalized boolean*.

key—a *designator* for a *function* of one argument, or **nil**.

result-list—a *list*.

Description:

union and **nunion** return a *list* that contains every element that occurs in either *list-1* or *list-2*.

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, `:test` or `:test-not` is used to determine whether they *satisfy the test*. The first argument to the `:test` or `:test-not` function is the part of the element of *list-1* extracted by the `:key` function (if supplied); the second argument is the part of the element of *list-2* extracted by the `:key` function (if supplied).

The argument to the `:key` function is an element of *list-1* or *list-2*; the return value is part of the supplied element. If `:key` is not supplied or **nil**, the element of *list-1* or *list-2* itself is supplied to the `:test` or `:test-not` function.

For every matching pair, one of the two elements of the pair will be in the result. Any element from either *list-1* or *list-2* that matches no element of the other will appear in the result.

If there is a duplication between *list-1* and *list-2*, only one of the duplicate instances will be in the result. If either *list-1* or *list-2* has duplicate entries within it, the redundant entries might or might not appear in the result.

The order of elements in the result do not have to reflect the ordering of *list-1* or *list-2* in any way. The result *list* may be **eq** to either *list-1* or *list-2* if appropriate.

union, nunion

Examples:

```
 (union '(a b c) '(f a d))
→ (A B C F D)
or→ (B C F A D)
or→ (D F A B C)
 (union '((x 5) (y 6)) '((z 2) (x 4)) :key #'car)
→ ((X 5) (Y 6) (Z 2))
or→ ((X 4) (Y 6) (Z 2))

 (setq lst1 (list 1 2 '(1 2) "a" "b")
       lst2 (list 2 3 '(2 3) "B" "C"))
→ (2 3 (2 3) "B" "C")
 (nunion lst1 lst2)
→ (1 (1 2) "a" "b" 2 3 (2 3) "B" "C")
or→ (1 2 (1 2) "a" "b" "C" "B" (2 3) 3)
```

Side Effects:

nunion is permitted to modify any part, *car* or *cdr*, of the *list structure* of *list-1* or *list-2*.

Exceptional Situations:

Should be prepared to signal an error of *type* **type-error** if *list-1* and *list-2* are not *proper lists*.

See Also:

intersection, Section 3.2.1 (Compiler Terminology), Section 3.6 (Traversal Rules and Side Effects)

Notes:

The `:test-not` parameter is deprecated.

Since the **nunion** side effect is not required, it should not be used in for-effect-only positions in portable code.