INDEX Simplified Common Lisp reference

Mathematics, Arithmetics, Logic and Comparisons

* + - / /= 1+ 1- < <= = > >= and ceiling cos decf eq eql equal equalp exp expt floor incf isqrt logand logior max min mod nil not or random round sin sqrt t zerop

Conses, Lists and related functions

append assoc butlast car cddr cdr cons consp first getf last list list-length listp mapc mapcan mapcar mapcon maplist member null pop push pushnew rest rplaca rplacd second set-difference union

Sequences (Lists, Strings) and Arrays

aref concatenate copy-seq count elt find length make-array make-sequence map map-into position reduce remove reverse search some string string-downcase string-upcase subseq vector vector-pop vector-push vector-push-extend

Symbol, Characters, Hash, Structure, Objects and Conversions

atom coerce gethash intern make-hash-table

Input and output

format read read-char read-line write-string

Functions, Evaluation, Flow Control, Definitions and Syntax

apply case cond defparameter defun eval flet funcall function if labels lambda let progn quote setf setq

Mathematics, Arithmetics, Logic and Comparisons

*

Syntax: Symbol type: function

`* numbers (zero or more) => number`

Argument description:

`numbers` numeric values

* function computes product of numbers. It does type conversions for numbers. There is no wraparound in integer numbers - they are arbitrary long. It works for all number types including integer, rational, floating point and complex.

```
(* 1 2 3) => 6
(* 1 2) => 2
(* 1) => 1
(*) => 1

(* 1234567890123456789 9876543210987654321) => 12193263113702179522374638011112635269
(* 1.3 -5) => -6.5
(* 1.3d0 -5) => -6.5d0
(* #c(2 4) 3) => #C(6 12)
(* 3/4 7/9) => 7/12
```

+

Syntax: Symbol type: function

`+ numbers (zero or more) => number`

Argument description:

`numbers` numeric values

+ function computes sum of numbers. It does type conversions for numbers. There is no wraparound in integer numbers - they are arbitrary long. It works for all number types including integer, rational, floating point and complex.

```
(+ 1 2 3) => 6
(+ 1 2) => 3
(+ 1) => 1
(+) => 0

(+ 1234567890123456789 9876543210987654321) => 11111111101111111110
(+ 1.3 -5) => -3.7
(+ 1.3d0 -5) => -3.7d0
```

```
(+ #c(2 4) 1) => #C(3 4)
(+ 3/4 7/9) => 55/36
```

-

Syntax: Symbol type: function

`-` *numbers* (one or more) => number

Argument description:

numbers numeric values

- function computes difference between first value and sum of the rest. When called with only one argument, it does negation. It does type conversions for numbers. There is no wraparound in integer numbers - they are arbitrary long. It works for all number types including integer, rational, floating point and complex.

```
(- 1 2 3) => -4
(- 1 2) => -1
(- 1) => -1

(- 1234567890123456789 9876543210987654321) => -8641975320864197532
(- 1.3 -5) => 6.3
(- 1.3d0 -5) => 6.3d0
(- #c(2 4) 1) => #C(1 4)
(- 3/4 7/9) => -1/36
```

/

Syntax: Symbol type: function

`/` *numbers* (one or more) => number

Argument description:

numbers numeric values

/ function computes division or reciprocal. When called with one argument it computes reciprocal. When called with two or more arguments it does compute division of the first by the all remaining number. It does type conversions for numbers. It works for all number types including integer, rational, floating point and complex. Note that division by zero invokes DIVISION-BY-ZERO condition.

```
(/ 10) => 1/10
(/ 10.0) => 0.1

(/ 10 2) => 5
(/ 2 10) => 1/5
(/ 100 2 5 2) => 5
(/ 100 (* 2 5 2)) => 5

(/ 1234567890123456789 9876543210987654321) => 13717421/109739369
(/ 1.3 -5) => -0.26
(/ 1.3d0 -5) => -0.26d0
(/ #c(2 4) 3) => #C(2/3 4/3)
(/ 3/4 7/9) => 27/28
```

/=

Syntax: Symbol type: function

`/=` *numbers* (one or more) => T or NIL

Argument description:

numbers numeric values

/= function compares numbers according to "equal" predicate. Result is true if no two numbers are equal to each other, otherwise result is false. Note that only two argument version result is negation of = function, that is (/= a b) is same as (not (= a b)).

```
(/= 1 2) => T
(/= 2 1) => T
(/= 2 2.001) => T
(/= 2 2) => NIL
(/= 2 2.0) => NIL
(/= 0.0 -0.0) => NIL
(/= #c(1.2 4.5) #c(1.2 4.5)) => NIL

(/= 1 2 3 4 5) => T
```

```
(/= 4 4 4 3 4) => NIL
(/= 1 2 3 4 4) => NIL
(/= 1 2 3 4.0 4) => NIL
(/= 5) => T
```

1+

Syntax: Symbol type: function

1+ *number* => number

Argument description:

number numeric value

1+ function adds one to the argument. See +.

```
(1+ 5) => 6
(1+ 1234567890123456789) => 1234567890123456790
(1+ 1.3) => 2.3
(1+ 1.3d0) => 2.3d0
(1+ #c(2 4)) => #C(3 4)
(1+ 3/4) => 7/4
```

1-

Syntax: Symbol type: function

1- *number* => number

Argument description:

number numeric value

1- function subtracts one from the argument. See -.

```
(1- 5) => 4
(1- 1234567890123456789) => 1234567890123456788
(1- 1.3) => 0.29999995
(1- 1.3d0) => 0.30000000000000004d0
(1- #c(2 4)) => #C(1 4)
(1- 3/4) => -1/4
```

<

Syntax: Symbol type: function

< *numbers* (one or more) => T or NIL

Argument description:

numbers numeric values

< function compares numbers according to "less than" predicate. Each (overlapping) pair of the numbers is compared by it. The result is true if all compared pairs satisfy comparison. Note that complex numbers cannot be compared.

```
(< 1 2) => T
(< 2 1) => NIL
(< 2 2.001) => T
(< 2 2) => NIL

(< 1234567890123456789 9876543210987654321) => T
(< 1 2 3 4 5) => T
(< 1 2 4 3 5) => NIL
(< 1 2 4 4 5) => NIL
(< 3/4 7/9) => T
(< 5) => T
```

<=

Syntax: Symbol type: function

<= *numbers* (one or more) => T or NIL

Argument description:

numbers numeric values

<= function compares numbers according to "less than or equal" predicate. Each (overlapping) pair of the numbers is compared by it. The result is true if all compared pairs satisfy comparison. Note that complex numbers cannot be compared.

```
(<= 1 2) => T
(<= 2 1) => NIL
(<= 2 2.001) => T
(<= 2 2) => T

(<= 1234567890123456789 9876543210987654321) => T
(<= 1 2 3 4 5) => T
(<= 1 2 4 3 5) => NIL
(<= 1 2 4 4 5) => T
(<= 3/4 7/9) => T
(<= 5) => T
```

=

Syntax: Symbol type: function

= *numbers* (one or more) => T or NIL

Argument description:

numbers numeric values

= function compares numbers according to "equal" predicate. Result is true if every specified number is equal to each other, otherwise result is false.

```
(= 1 2) => NIL
(= 2 1) => NIL
(= 2 2.001) => NIL
(= 2 2) => T
(= 2 2.0) => T
(= 0.0 -0.0) => T
(= #c(1.2 4.5) #c(1.2 4.5)) => T

(= 1 2 3 4 5) => NIL
(= 4 4 4 3 4) => NIL
(= 4 4 4 4 4) => T
(= 4 4 4 4.0 4) => T
(= 5) => T
```

>

Syntax: Symbol type: function

> *numbers* (one or more) => T or NIL

Argument description:

numbers numeric values

> function compares numbers according to "greater than" predicate. Each (overlapping) pair of the numbers is compared by it. The result is true if all compared pairs satisfy comparison. Note that complex numbers cannot be compared.

```
(> 2 1) => T
(> 1 2) => NIL
(> 2.001 2) => T
(> 2 2) => NIL

(> 9876543210987654321 1234567890123456789) => T
(> 5 4 3 2 1) => T
(> 5 3 4 2 1) => NIL
(> 5 4 4 2 1) => NIL
(> 7/9 3/4) => T
(> 5) => T
```

>=

Syntax: Symbol type: function

>= *numbers* (one or more) => T or NIL

Argument description:

numbers numeric values

>= function compares numbers according to "greater than or equal" predicate. Each (overlapping)

pair of the numbers is compared by it. The result is true if all compared pairs satisfy comparison. Note that complex numbers cannot be compared.

```
(>= 2 1) => T
(>= 1 2) => NIL
(>= 2.001 2) => T
(>= 2 2) => T

(>= 9876543210987654321 1234567890123456789) => T
(>= 5 4 3 2 1) => T
(>= 5 3 4 2 1) => NIL
(>= 5 4 4 2 1) => T
(>= 7/9 3/4) => T
(>= 5) => T
```

and

Syntax: Symbol type: macro

and *forms* (zero or more) => value

Argument description:

forms	forms which value is considered

AND macro computes logical "and" function. Forms evaluation starts from left. Value from the first form that decides result is returned so forms at end of argument list may not evaluated.

```
(and t t t nil t) => NIL
(and t t t t) => T
(and) => T

(and (progn (write "SEEN") nil) (progn (write "UNSEEN") t)) => "SEEN" NIL
(and 4 5 6) => 6
```

ceiling

Syntax: Symbol type: function

ceiling *number divisor* => quotient (numeric value), remainder (numeric value)

Argument description:

number	number
divisor	non-zero real number, default is 1

CEILING function returns two values, the first is result of dividing number by divisor and truncating toward positive infinity. Second result remainder that satisfies equation: quotient * divisor + remainder = number.

```
(ceiling 10) => 10, 0
(ceiling 10.3 2) => 6, -1.6999998
(ceiling 3/4) => 1, -1/4

(multiple-value-list (ceiling 20 7)) => (3 -1)
```

cos

Syntax: Symbol type: function

cos *number* => numeric value

Argument description:

number	numeric value, angle in radians

COS function computes cosine of value in radians.

```
(cos 0.0) => 1.0
(cos 1.0) => 0.5403023
(cos 1.0d0) => 0.5403023058681398d0
(cos #c(1.0 1.0)) => #C(0.83373 -0.9888977)
```

decf

Syntax: Symbol type: macro

decf *place* *decrement* (optional) => numeric value

Argument description:

place	place with numeric value
decrement	numeric value

DECF macro modifies a place with numeric value. Its value is decremented by decrement number. Default decrement is 1.

```
(let ((a 10)) (decf a) a) => 9
(let ((a 10)) (decf a 2.3) a) => 7.7
(let ((a 10)) (decf a -2.3) a) => 12.3
(let ((a (list 10 11 12 13))) (decf (elt a 2) 2.3) a) => (10 11 9.7 13)
```

eq

Syntax: Symbol type: function

eq *object1* *object2* => T or NIL

Argument description:

object1	first object
object2	second object

EQ function compares object identity. It works for symbols and identical objects. It is not suitable for comparing numbers - see EQL and =. Result is true if they are same, otherwise false.

```
(eq 'moo 'moo) => T
(eq 'moo 'foo) => NIL
(eq 1 1) => T
(eq 1 2) => NIL
(eq 1234567890123456789 1234567890123456789) => NIL

(eq (cons 1 2) (cons 1 2)) => NIL
(let ((x (cons 1 2))) (eq x x)) => T
```

eql

Syntax: Symbol type: function

eql *object1* *object2* => T or NIL

Argument description:

object1	first object
object2	second object

EQL function compares object identity, numbers and characters. Numbers are considered as equal only when they have the both same value and type. Result is true if they are same, otherwise false.

```
(eql 'moo 'moo) => T
(eql 'moo 'foo) => NIL
(eql 1 1) => T
(eql 1 2) => NIL
(eql 1234567890123456789 1234567890123456789) => T

(eql 1.0 1) => NIL
(eql 1.0 1.0) => T

(eql (cons 1 2) (cons 1 2)) => NIL
(let ((x (cons 1 2))) (eql x x)) => T
```

equal

Syntax: Symbol type: function

equal *object1* *object2* => T or NIL

Argument description:

object1	first object
object2	second object

EQUAL function compares same things as eql, additionally result is true under some other situations: conses are compared recursively (in both car and cdr part), string and bit-vectors are compared element-wise. Result is true if they are same, otherwise false.

```
(equal "moo" "moo") => T
(equal "moo" "MoO") => NIL
(equal #*1010101 #*1010101) = T
(equal (vector 2 3 4) (vector 2 3 4)) = NIL
(equal (cons 1 2) (cons 1 2)) => T
(let ((x (cons 1 2))) (equal x x)) => T

(equal 'moo 'moo) => T
(equal 'moo 'foo) => NIL
(equal 1 1) => T
(equal 1 2) => NIL
(equal 1234567890123456789 1234567890123456789) => T

(equal 1.0 1) => NIL
(equal 1.0 1.0) => T
```

equalp

Syntax: Symbol type: function

equalp *object1* *object2* => T or NIL

Argument description:

object1	first object
object2	second object

EQUALP function compares same things as equal, additionally result is true under some other situations: conses are compared recursively (in both car and cdr part), any sequence is compared recursively (element-wise), strings and characters are compared case insensitively. Result is true if they are same, otherwise false.

```
(equalp "moo" "moo") => T
(equalp "moo" "MoO") => T
(equalp "moo" "moo ") => NIL
(equalp (vector 2 3 4) (vector 2 3 4)) = T
(equalp (cons 1 2) (cons 1 2)) => T
(let ((x (cons 1 2))) (equalp x x)) => T

(equalp 'moo 'moo) => T
(equalp 'moo 'foo) => NIL
(equalp "a" 'a) => NIL
(equalp 1 1) => T
(equalp 1 2) => NIL
(equalp 1234567890123456789 1234567890123456789) => T

(equalp 1.0 1) => T
(equalp 1.0 1.0) => T
```

exp

Syntax: Symbol type: function

exp *number* => number

Argument description:

number	number to be raised

EXP function returns e raised to the power number, where e is the base of the natural logarithms.

```
(exp 1) => 2.7182817
(exp 1.0) => 2.7182817
(exp 1.0d0) => 2.718281828459045d0
(exp 10) => 22026.465

(log (exp 10)) => 10.0
```

expt

Syntax: Symbol type: function

expt *base-number* *power-number* => number

Argument description:

base-number	number to be raised
power-number	number that specifies power

EXPT function returns base-number raised to the power-number.

```
(expt 2 8) => 256
(expt 2 32) => 4294967296
(expt 2 64) => 18446744073709551616
(expt 10 3) => 1000
(expt 5 1/3) => 1.709976
(expt 1.709976 3) => 5.0

(expt 1 2) => 1
(expt 1.0 2) => 1.0
(expt 1.0d0 2) => 1.0d0
(expt -2 5) => -32
(expt 2 2.5) => 5.656854
(expt -2 2.5) => #C(1.7318549e-15 5.656854)
(expt 2 -3) => 1/8
(expt -2 -3) => -1/8
(expt -2.4 -3) => -0.072337955
(expt -2.4 -3.3) => #C(-0.032698035 0.045004968)

(expt (expt 10 5) 1/5) => 10.0
```

floor

Syntax: Symbol type: function

floor *number divisor* => quotient (numeric value), remainder (numeric value)

Argument description:

number	number
divisor	non-zero real number, default is 1

FLOOR function returns two values, the first is result of dividing number by divisor and truncating toward negative infinity. Second result remainder that satisfies equation: quotient * divisor + remainder = number.

```
(floor 10) => 10, 0
(floor 10.3 2) => 5, 0.3000002
(floor 3/4) => 0, 3/4

(multiple-value-list (floor 20 7)) => (2 6)
```

incf

Syntax: Symbol type: macro

incf *place increment* (optional) => numeric value

Argument description:

place	place with numeric value
increment	numeric value

INCF macro modifies a place with numeric value. Its value is incremented by increment number. Default increment is 1.

```
(let ((a 10)) (incf a) a) => 11
(let ((a 10)) (incf a 2.3) a) => 12.3
(let ((a 10)) (incf a -2.3) a) => 7.7
(let ((a (list 10 11 12 13))) (incf (elt a 2) 2.3) a) => (10 11 14.3 13)
```

isqrt

Syntax: Symbol type: function

isqrt *number* => integer value

Argument description:

number	positive integer

ISQRT function computes integer part of square root of number. See also SQRT.

```
(isqrt 10) => 3
(isqrt 4) => 2
```

logand

Syntax: Symbol type: function

logand *integers* (zero or more) => integer

Argument description:

integers integers for bitwise operations

LOGIOR function computes bitwise logical "and" function.

```
(logand) => -1
(logand 1 2) => 0
(logand #xff #xaa) => 170
(logand #b1010 #b100 #b11) => 0
```

logior

Syntax: Symbol type: function

logior *integers* (zero or more) => integer

Argument description:

integers integers for bitwise operations

LOGIOR function computes bitwise logical "or" function.

```
(logior) => 0
(logior 1 2) => 3
(logior #xf0 #xf) => 255
(logior #b1010 #b100 #b11) => 15
```

max

Syntax: Symbol type: function

max *numbers* (one or more) => numeric value

Argument description:

numbers comparable numbers

MAX function returns the maximal number from arguments. Type of resulting number may be different when arguments multiple precision numbers.

```
(max 1 3 2) => 3
(max 4) => 4
(= 3.0 (max 3 1 3.0 2 3.0d0)) => T
```

min

Syntax: Symbol type: function

min *numbers* (one or more) => numeric value

Argument description:

numbers comparable numbers

MIN function returns the minimal number from arguments. Type of resulting number may be different when arguments multiple precision numbers.

```
(min 1 3 2) => 1
(min 4) => 4
(= 3.0 (min 3 7 3.0 8 3.0d0)) => T
```

mod

Syntax: Symbol type: function

mod *number divisor* => number

Argument description:

number real number

divisor real number

MOD function returns modulus of two integer arguments. Non-integer arguments are first turned into integers by floor operation. Note that division by zero invokes DIVISION-BY-ZERO condition.

```
(mod -1 5) => 4
(mod 13 4) => 1
(mod -13 4) => 3
(mod 13 -4) => -3
(mod -13 -4) => -1
(mod 13.4 1) => 0.4
(mod -13.4 1) => 0.6
```

nil

Syntax: Symbol type: symbol

nil => symbol

NIL symbol denotes empty list and false value. There is also ubiquitous NIL constant which contains NIL symbol. NIL is considered as false value by comparison functions and control operators (unlike any other). Empty list, that is '() or even (), is the same as NIL value.

```
nil => NIL
'nil => NIL
() => NIL
'() => NIL
```

```
(eq nil 'nil) => T
(eq 'nil ()) => T
(eq () '()) => T
```

```
(not t) => NIL
(not nil) => T
(not 234.3) => NIL
```

not

Syntax: Symbol type: function

not *value* => value

Argument description:

value logical value

NOT computes logical negation. Note that any other value than NIL is considered as true. NOT is identical to NULL, but used in conjunction with boolean logic processing unlike NOT which is used in list processing.

```
(not t) => NIL
(not nil) => T
(not 234.3) => NIL
```

or

Syntax: Symbol type: macro

or *forms* (zero or more) => value

Argument description:

forms forms which value is considered

OR macro computes logical "or" function. Forms evaluation starts from left. Value from the first form that decides result is returned so forms at end of argument list may not evaluated.

```
(or t t t nil t) => T
(or nil nil nil) => NIL
(or) => NIL
```

```
(or (progn (write "SEEN") 123) (progn (write "UNSEEN") t)) => "SEEN" 123
(or 4 5 6) => 4
```

random

Syntax: Symbol type: function

random *limit* *random-state* (optional) => numeric value

Argument description:

limit	positive number, integer or real
random-state	object representing random state

RANDOM function generates random numbers. For integer argument N, result is integer between zero (including) and N (excluding). For real argument X, result is real between zero (including) and X (excluding). All generated numbers have approximately same probability. Default value for random-state is stored in *random-state* global variable.

```
(<= 0 (random 20) 19) => T
(let ((x (random 1.0))) (or (= x 0) (< 0 x 1.0))) => T
```

round

Syntax: Symbol type: function

round *number* *divisor* => quotient (numeric value), remainder (numeric value)

Argument description:

number	number
divisor	non-zero real number, default is 1

FLOOR function returns two values, the first is result of dividing number by divisor and truncating toward nearest even integer. Second result remainder that satisfies equation: quotient * divisor + remainder = number.

```
(round 10) => 10, 0
(round 10.3 2) => 5, 0.3000002
(round 3/4) => 0, 3/4
(round 3/2) => 2, -1/2

(multiple-value-list (round 20 7)) => (3 -1)
```

sin

Syntax: Symbol type: function

sin *number* => numeric value

Argument description:

number	numeric value, angle in radians

SIN function computes sine of value in radians.

```
(sin 0.0) => 0.0
(sin 1.0) => 0.84147096
(sin 1.0d0) => 0.8414709848078965d0
(sin #c(1.0 1.0)) => #C(1.2984576 0.63496387)
```

sqrt

Syntax: Symbol type: function

sqrt *number* => numeric value

Argument description:

number	number

SQRT function computes square root of number. Number may be integer, real or complex. See also ISQRT.

```
(sqrt 10) => 3.1622777
(sqrt 10.0) => 3.1622777
(sqrt 10.0d0) => 3.1622776601683795d0
(sqrt 4) => 2.0

(sqrt -4) => #C(0.0 2.0)
(sqrt #C(0.0 2.0)) => #C(1.0 1.0)
```

t

Syntax: Symbol type: symbol

t => symbol

T symbol denotes true value. There is also ubiquitous T constant which contains T symbol. T is not only true value, all values except NIL are treat as true by comparison functions and control operators. Note that constants cannot be redefined (even locally) so there is no chance to make variable t in same name space with T (the true symbol).

```
't => T
t => T
(eq 't t) => T

(not t) => NIL
(not nil) => T
(not 234.3) => NIL
```

zerop

Syntax: Symbol type: function

zerop *number* => boolean

Argument description:

number a number

ZEROP function returns true if the argument is zero.

```
(zerop 0) => T
(zerop -0.0) => T
(zerop #c(0 0.0)) => T
(zerop #c(0 0.1)) => NIL
(zerop 3/3) => NIL
```

Conses, Lists and related functions

append

Syntax: Symbol type: function

append *list* (zero or more) => list

Argument description:

list lists to be concatenated

APPEND function concatenates list arguments into one list. Resulting list is shallow copy of specified lists except for the last which is directly shared. See also MAPCAN, CONS, LIST, LIST*.

```
(append) => NIL
(append '(1 2 3)) => (1 2 3)
(append '(1 2 3) '(4 5 6)) => (1 2 3 4 5 6)
(append '(1 2 3) '(4 5 6) '(7 8 9)) => (1 2 3 4 5 6 7 8 9)
(let ((x '(tail list))) (eq x (cddr (append '(front list) x)))) => T
```

assoc

Syntax: Symbol type: function

assoc *item* *alist* *key* (keyword) *test* (keyword) => cons cell or NIL

Argument description:

item a key object

alist alist - list of cons cell with key-value pairs

key function for extracting key before test

test function key and item comparison

ASSOC function searches supplied list for cons cell that have item as car part. Return value is the cell with key-value pair which key matched testing conditions, otherwise NIL. Default comparison operator is EQL.

Associative list, or for short alist, is a list with key-value pairs in cons cells. That is ((key1 . value1) (key2 . value2) ...)

```
(assoc 'a '((a . 1) (b . 2) (c . 3))) => (A . 1)
(assoc 'x '((a . 1) (b . 2) (c . 3))) => NIL
(assoc 'b '((a . 1) (b . 2) (c . 3) (b . 4))) => (B . 2)
(assoc "b" '(("a" . 1) ("b" . 2))) => NIL
(assoc "b" '(("a" . 1) ("b" . 2)) :test #'equal) => ("b" . 2)
(assoc 7 '((6 . a) (9 . b)) :key #'1+) => (6 . A)
(assoc 5 nil) => NIL
```

butlast

Syntax: Symbol type: function

butlast *list* *n* (optional) => list

Argument description:

list a list

n a non-negative integer, default is 1

BUTLAST function returns the argument list copy without N last elements. See LAST.

```
(butlast '(1 2 3)) => (1 2)
```

```
(butlast '(1 2 3) 0) => (1 2 3)
(butlast '(1 2 3) 1) => (1 2)
(butlast '(1 2 3) 2) => (1)
(butlast '(1 2 3) 3) => NIL
(butlast '(1 2 3) 4) => NIL
```

car

Syntax: Symbol type: function

car *list* => value

Argument description:

list cons or full list

CAR function returns the first element of list, that is the car part of its cons cell argument. CAR is identical to FIRST.

```
(car '(1 2 3)) => 1
(car (cons 'a 'b)) => A
(car (cons '(1 2 3) '(a b c))) => (1 2 3)
(car '()) => NIL
(car nil) => NIL
```

cddr

Syntax: Symbol type: function

cddr *list* => value

Argument description:

list cons or full list

CDDR function is composition of two CDR functions. That is, (CDDR X) is same as (CDR (CDR X)). There are other CAR and CDR combinations, see HyperSpec, CAR and CDR.

```
(cddr '(1 2 3 4)) => (3 4)
(cddr '(1 2 . x)) => X
(cddr '(1 . nil)) => NIL
(cddr nil) => NIL
```

cdr

Syntax: Symbol type: function

cdr *list* => value

Argument description:

list cons or full list

CDR function returns cdr part of cell in the argument, that is list of all elements but first. CDR is identical to REST.

```
(cdr '(1 2 3)) => (2 3)
(cdr (cons 'a 'b)) => B
(cdr (cons '(1 2 3) '(a b c))) => (A B C)
(cdr '()) => NIL
(cdr nil) => NIL
```

cons

Syntax: Symbol type: function

cons *car-part* *car-part* => cons cell

Argument description:

car-part an object

car-part an object

CONS function make new cons object. The cons cell contains exactly two values. The first is named car, the second is named cdr. These cells are used to create one-way linked lists. See also CAR, CDR and LIST.

These names come from historical names "Contents of Address part of Register" and "Contents of Decrement part of Register".

```
(cons 1 2) => (1 . 2)
(cons 1 (cons 2 (cons 3 nil))) => (1 2 3)
(cons 1 (cons 2 (cons 3 'x))) => (1 2 3 . X)
(cons (cons (cons 'a 'b) 'c) 'd) => (((A . B) . C) . D)
(car (cons 1 2)) => 1
(cdr (cons 1 2)) => 2
```

consp

Syntax: Symbol type: function

consp *object* => T or NIL

Argument description:

object an object

CONSP function returns true if the argument refers to cons cell, otherwise it returns false. See CONS and LIST.

```
(consp nil) => NIL
(consp 'some-symbol) => NIL
(consp 3) => NIL
(consp "moo") => NIL
(consp (cons 1 2)) => T
(consp '(1 . 2)) => T
(consp '(1 2 3 4)) => T
(consp (list 1 2 3 4)) => T
```

first

Syntax: Symbol type: function

first *list* => value

Argument description:

list cons or full list

FIRST function returns the first element of list, that is the car part of its cons cell argument. FIRST is identical to CAR.

```
(first '(1 2 3)) => 1
(first (cons 'a 'b)) => A
(first (cons '(1 2 3) '(a b c))) => (1 2 3)
(first '()) => NIL
(first nil) => NIL
```

getf

Symbol type: function

getf *place key default* (optional) => value

Argument description:

place	a place with list
key	keying value, also know as indicator
default	answer when key-value pair is not found, default is NIL

GETF function searches supplied plist for value with matching key. Plist is list of even number of items. Each item pair specifies key and value. I.e. (K1 V1 K2 V2 ...). Return value is either value for first matching key, or specified default. Keys are matched by EQ function, therefore only suitable values are symbols and integers in range between MOST-NEGATIVE-FIXNUM and MOST-POSITIVE-FIXNUM constants. See also SETF, ASSOC and FIND.

```
(getf '(a b 4 d a x) 'a) => B
(getf '(a b 4 d a x) 'x) => NIL
(getf '(a b 4 d a x) 'x 'not-found) => NOT-FOUND
(getf '(a b 4 d a x) 4 'not-found) => D
```

last

Syntax: Symbol type: function

last *list n* (optional) => list

Argument description:

list	a list
n	a non-negative integer, default is 1

LAST function returns the list of N last elements of list argument. See BUTLAST.

```
(last '(1 2 3)) => (3)
```

```
(last '(1 2 3) 0) => NIL
(last '(1 2 3) 1) => (3)
(last '(1 2 3) 2) => (2 3)
(last '(1 2 3) 3) => (1 2 3)
(last '(1 2 3) 4) => (1 2 3)
(last '(a . b) 0) => B
(last '(a . b) 1) => (A . B)
(last '(a . b) 2) => (A . B)
```

list

Syntax: Symbol type: function

list *list* (zero or more) => list

Argument description:

list	list of objects

LIST function makes new list from arguments.

```
(list 1 2 3) => (1 2 3)
(list 'a #c(1 2) "moo") => (A #C(1 2) "moo")
(car (list 1 2 3)) => 1
(cdr (list 1 2 3)) => (2 3)
(list) => NIL
(eq (list) nil) => T
(eq (list) '()) => T
(equal (list 1) (cons 1 nil)) => T
(equal (list 1 'a) (cons 1 (cons 'a nil))) => T
(equal (list 1 'a 3) (cons 1 (cons 'a (cons 3 nil)))) => T
(equal (list 1 'a 3) '(1 . (a . (3 . nil)))) => T
(equal '(1 2 3) (list 1 2 3)) => T
```

list-length

Syntax: Symbol type: function

list-length *list* => integer or NIL

Argument description:

list	list or cyclic list

LIST-LENGTH function computes length of the lists. LIST-LENGTH will return NIL if it encounters cyclic cons cell structure. LIST-LENGTH is slower than LENGTH because of additional cycle checking.

```
(list-length '(a . (b . nil))) => 2
(list-length '#1=(a . (b . #1#))) => NIL

(list-length (list 'a 'b 'c)) => 3
(list-length nil) => 0
(list-length (cons "moo" nil)) => 1
(list-length (cons "moo" (cons "boo" nil))) => 2
```

listp

Syntax: Symbol type: function

listp *object* => T or NIL

Argument description:

object	an object

LISTP function returns true if the argument is refers to object of type list; otherwise it returns false. Objects of list type can contain cons cells or NIL value (list terminator). See CONS, CONSP and LIST.

```
(listp nil) => T
(listp 'some-symbol) => NIL
(listp 3) => NIL
(listp "moo") => NIL
(listp (cons 1 2)) => T
(listp '(1 . 2)) => T
(listp '(1 2 3 4)) => T
(listp (list 1 2 3 4)) => T
```

mapc

Syntax: Symbol type: function

mapc *fn* *lists* (one or more) => the first list from lists argument

Argument description:

fn	function that takes as many arguments as there are lists
lists	lists which elements are processed in parallel

MAPC applies function FN to elements of lists with same index. Each application result forgotten. Elemnts are processed only up to length of the shortest list argument. See MAPCAR, MAPCAN, MAPCON, DOLIST.

```
(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)
```

mapcan

Syntax: Symbol type: function

mapcan *fn* *lists* (one or more) => list

Argument description:

fn	function that takes as many arguments as there are lists
lists	lists which elements are processed in parallel

MAPCAN applies function FN to elements of lists with same index. Each application result is concatenated into resulting list. See MAPCAR.

```
(mapcan (lambda (x) (list (+ x 10) 'x)) '(1 2 3 4)) => (11 X 12 X 13 X 14 X)

(mapcan #'list '(a b c d)) => (A B C D)

(mapcan (lambda (x) (if (> x 0) (list x) nil)) '(-4 6 -23 1 0 12 )) => (6 1 12)
```

mapcar

Syntax: Symbol type: function

mapcar *fn lists* (one or more) => list

Argument description:

fn	function that takes as many arguments as there are lists
lists	lists which elements are processed in parallel

MAPCAR applies function FN to elements of lists with same index. Each application result is put into resulting list. Length of resulting list is the length of the shortest list argument. See MAPCAN.

```
(mapcar (lambda (x) (+ x 10)) '(1 2 3 4)) => (11 12 13 14)

(mapcar #'round '(1.3 2.7 3.4 4.5)) => (1 3 3 4)

(mapcar #'list '(123 symbol "string" 345) '(1 2 3)) => ((123 1) (SYMBOL 2) ("string" 3))

(mapcar #'* '(3 4 5) '(4 5 6)) => (12 20 30)
```

mapcon

Syntax: Symbol type: function

mapcon *fn lists* (one or more) => list

Argument description:

fn	function that takes as many arguments as there are lists
lists	lists which elements are processed in parallel

MAPCON applies function FN to the successive cdr of lists. Each application result is DESTRUCTIVELY concatenated into resulting list. In case of FN results that are fresh lists (non-sharing), the result is same as with (APPLY #'APPEND (MAPLIST ...)). See MAPCAR, MAPCAN, MAPCON, MAP, MAPC.

```
(mapcon (lambda (x) (list 'start x 'end)) '(1 2 3 4))
=> (START (1 2 3 4) END START (2 3 4) END START (3 4) END START (4) END)
```

maplist

Syntax: Symbol type: function

maplist *fn lists* (one or more) => list

Argument description:

fn	function that takes as many arguments as there are lists
lists	lists which elements are processed in parallel

MAPLIST applies function FN to the successive cdr of lists. Each application result is concatenated into resulting list. See MAPCAR, MAPCAN, MAPCON, MAP, MAPC.

```
(maplist (lambda (x) (list 'start x 'end)) '(1 2 3 4))
=> ((START (1 2 3 4) END) (START (2 3 4) END) (START (3 4) END) (START (4) END))
```

member

Syntax: Symbol type: function

member *item list test* (keyword) *key* (keyword) => tail or NIL

Argument description:

item	an item to be found
list	a list to be searched
test	function key and item comparison
key	function for extracting value before test

MEMBER function searches a list for the first occurrence of an element (item) satisfying the test. Return value is tail of the list starting from found element or NIL when item is not found. See also MEMBER-IF, POSITION, POSITION-IF, FIND and FIND-IF.

```
(member 1 '(0 1 0 0 0 1 0)) => (1 0 0 0 1 0)
(member 2 '(0 1 0 0 0 1 0)) => NIL
(member #\h '(#\H #\o #\l #\a)) => NIL
(member #\h '(#\H #\o #\l #\a) :test #'char-equal) => (#\H #\o #\l #\a)
(member #\h '(#\H #\o #\l #\a) :key #'char-downcase) => (#\H #\o #\l #\a)
```

null

Syntax: Symbol type: function

null *object* => T or NIL

Argument description:

object an object

NULL function returns true if the argument is NIL, otherwise it returns false. NULL is identical to NOT, but used in conjunction with list processing unlike NOT which is used in boolean logic processing.

```
(null '()) => T
(null '(1 2 3)) => NIL
(null nil) => T
(null t) => NIL
(null 234.4) => NIL
(null "lisp") => NIL
```

pop

Syntax: Symbol type: macro

pop *place* => list

Argument description:

place a place containing list

POP macro modifies variable or generally place. It replaces the cons cell value with its cdr. Effectively it removes first element of the list found at the place. Result is the first element of the original list. See also PUSH, PUSH-NEW and ACONS.

```
(let ((x '(1 2 3))) (pop x)) => 1
(let ((x '(1 2 3))) (pop x) x) => (2 3)
(let ((x '((a b c) (3 2 1) (e f g)))) (pop (second x)) x) => ((A B C) (2 1) (E F G))
(let ((x '(()))) (pop x) x) => NIL
```

push

Syntax: Symbol type: macro

push *item* *place* => list

Argument description:

item an object

place a place which can contain any object, but usually list

PUSH macro modifies variable or generally place. It makes a new cons cell filled with item as car and previous value as cdr, that is effectively prepends new item to list found at the place. See also PUSH-NEW, ACONS and POP.

```
(let ((x 'x)) (push 4 x) x) => (4 . X)
(let ((x '(3 2 1))) (push 4 x) x) => (4 3 2 1)
(let ((x '((a b c) (3 2 1) (e f g)))) (push 4 (second x)) x) => ((A B C) (4 3 2 1) (E F G))
```

pushnew

Syntax: Symbol type: macro

pushnew *item* *place* *key* (keyword) *test* (keyword) => list

Argument description:

item	an object
place	a place which can contain any object, but usually list
key	function for extracting value before test
test	function key and item comparison

PUSHNEW macro modifies variable or generally place. It conditionally makes a new cons cell filled with item as car and previous value as cdr, that is effectively prepends new item to list found at the place. New element is pushed only when it does not appear in place. Test argument specifies comparison operator. Default comparison operator is EQL. Key argument specifies function for extracting relevant value from list items. Default key is IDENTITY. See also PUSH-NEW, ACONS and POP.

```
(let ((x 'x)) (pushnew 4 x) x) => (4 . X)
(let ((x '(3 2 1))) (pushnew 4 x) x) => (4 3 2 1)
(let ((x '(3 2 1))) (pushnew 3 x) x) => (3 2 1)
(let ((x '((a b c) (3 2 1) (e f g)))) (pushnew 4 (second x)) x) => ((A B C) (4 3 2 1) (E F G))
(let ((x '((a b c) (3 2 1) (e f g)))) (pushnew 3 (second x)) x) => ((A B C) (3 2 1) (E F G))
(let ((x '("3" "2" "1"))) (pushnew "3" x) x) => (3 2 1)
(let ((x '("31" "24" "13"))) (pushnew "44" x :key (lambda (x) (elt x 0))) x) => ("44" "31" "24" "13")
(let ((x '("31" "24" "13"))) (pushnew "44" x :key (lambda (x) (elt x 1))) x) => ("31" "24" "13")
```

rest

Syntax: Symbol type: function

rest *list* => value

Argument description:

list	cons or full list

REST function returns list of all elements but first, that is cdr part of argument. REST is identical to CDR.

```
(rest '(1 2 3)) => (2 3)
(rest (cons 'a 'b)) => B
(rest (cons '(1 2 3) '(a b c))) => (A B C)
(rest '()) => NIL
(rest nil) => NIL
```

rplaca

Syntax: Symbol type: function

rplaca *cons* *object* => cons

Argument description:

cons	a cons cell
object	an object

RPLACA function changes CAR part of CONS cell to specified value. See RPLACD, SETF, CONS.

This can be also writen as (SETF (CAR cons) object).

```
(let ((my-list (list 5 3 6 2))) (rplaca my-list 'bla) my-list) => (BLA 3 6 2)
```

rplacd

Syntax: Symbol type: function

rplacd *cons* *object* => cons

Argument description:

cons	a cons cell
object	an object

RPLACD function changes CDR part of CONS cell to specified value. See RPLACA, SETF, CONS.

This can be also writen as (SETF (CDR cons) object).

```
(let ((my-list (list 5 3 6 2))) (rplaca (cdr my-list) '(x y)) my-list) => (5 (X Y) 6 2)
```

second

Syntax: Symbol type: function

second *list* => value

Argument description:

list cons or full list

SECOND function returns second element of list, that is car part of cdr part of its cons cell. SECOND is identical to CADR.

```
(second '(1 2 3)) => 2
(second (cons 'a (cons 'b 'c))) => B
```

set-difference

Syntax: Symbol type: function

set-difference *list1* *list2* *key* (keyword) *test* (keyword) => list

Argument description:

list1 a list
list2 a list
key function for extracting value before test
test function key and item comparison

SET-DIFFERENCE function computes set difference, that is a list of elements that appear in list1 but do not appear in list2. Test argument specifies comparison operator. Default comparison operator is EQL. Key argument specifies function for extracting relevant value from list items. Default key is IDENTITY. Resulting item order is not specified. See also SET-EXCLUSIVE-OR, UNION and INTERSECTION.

```
(set-difference '(a b c) '(b c d)) => (A)
(set-difference '("a" "b" "c") '("b" "c" "d")) => ("c" "b" "a")
(set-difference '("a" "b" "c") '("b" "c" "d") :test #'equal) => ("a")
(set-difference '((a . 2) (b . 3) (c . 1)) '((b . 1) (c . 2) (d . 4)) :test #'equal) => ((C . 1) (B . 3) (A . 2))
(set-difference '((a . 2) (b . 3) (c . 1)) '((b . 1) (c . 2) (d . 4)) :key #'car) => ((A . 2))
(set-difference '((a . 2) (b . 3) (c . 1)) '((b . 1) (c . 2) (d . 4)) :key #'cdr) => ((B . 3))
```

union

Syntax: Symbol type: function

union *list-1* *list-2* *key* (keyword) *test* (keyword) *test-not* (keyword) => list

Argument description:

list-1 list to be joined
list-2 other list to be joined
key function for extracting value before test
test function for comparison of two values
test-not function for comparison of two values

UNION function computes union of two lists. Resulting list contains elements that appear in one or other list. See INTERSECTION, SET-DIFFERENCE, SET-EXCLUSIVE-OR.

```
(union '(1 2 3) '(2 3 4)) => (1 2 3 4)
(union '((1) (2) (3)) '((2) (3) (4))) => ((3) (2) (1) (2) (3) (4))
(union '((1) (2) (3)) '((2) (3) (4)) :test #'equal) => ((1) (2) (3) (4))
(union '((1) (2) (3)) '((2) (3) (4)) :key #'first) => ((1) (2) (3) (4))
```

Sequences (Lists, Strings) and Arrays

aref

Syntax: Symbol type: function

aref *array* *subscripts* (zero or more) => element

Argument description:

array	an array
subscripts	a list of valid array indices

AREF function accesses specified elements of arrays. Every array index is counted from zero. Accessing out-of-bounds indices signals condition, or causes crash and/or undefined behavior, depending on compilation safety mode. Note that vectors (including strings which are special vectors) are treated as one dimensional arrays so aref works on them too.

AREF with conjunction of SETF may be used to set array elements.

```
(aref "hola" 0) => #\h
(aref "hola" 3) => #\a
(aref #(5 3 6 8) 1) => 3
(aref (make-array '(10 10) :initial-element 'moo) 9 9) => MOO

(let ((a (make-array '(3 3) :initial-element 'moo))) (setf (aref a 1 1) 'x) a) => #2A((MOO MOO MOO) (MOO X MOO) (MOO MOO MOO))
```

concatenate

Syntax: Symbol type: function

concatenate *result-type* *seqs* (one or more) => sequence

Argument description:

result-type	sequence type specifier or NIL
seqs	sequences

CONCATENATE creates new sequence and fills it with data from arguments. See also MAPCAN.

```
(concatenate 'string "hello" " " "world") => "hello world"
(concatenate 'list "hello" " " "world") => (#\h #\e #\l #\l #\o #\  #\w #\o #\r #\l #\d)
(concatenate 'vector "hello" " " "world") => #(#\h #\e #\l #\l #\o #\  #\w #\o #\r #\l #\d)
(concatenate 'vector '(1 2) '(3 4)) => #(1 2 3 4)
```

copy-seq

Syntax: Symbol type: function

copy-seq *seq* => sequence

Argument description:

seq	a sequence

COPY-SEQ function makes new sequence copy from old sequence. Note that there is no COPY-ARRAY function, but it can be emulated by this tricky code bellow:

```
(defun copy-array (array)
 (let ((dims (array-dimensions array)))
   (adjust-array
    (make-array dims :displaced-to array)
    dims)))

(let ((a "hello world")) (eq a (copy-seq a))) => NIL
(let ((a "hello world")) (equal a (copy-seq a))) => T
```

count

Syntax: Symbol type: function

count *item* *sequence* *test* (keyword) *from-end* (keyword) *start* (keyword) *end* (keyword) *key* (keyword) *test* (keyword) *test-not* (keyword) => integer

Argument description:

item	an item to be found
sequence	a sequence to be searched
test	function key and item comparison
from-end	direction of search, default is NIL - forward
start	starting position for search, default is 0

end	final position for search, default is NIL - end of sequence
key	function for extracting value before test
test	function for comparison of two values
test-not	function for comparison of two values

COUNT function counts specified elements in sequence. Return value is number of occurancesf or NIL if no occurance is not found. See also COUNT-IF, POSITION, POSITION-IF, FIND, FIND-IF and MEMBER.

```
(count #\s "Some sequence") => 1
(count #\s "Some sequence" :key #'char-downcase) => 2
(count #\s "Some sequence" :key #'char-downcase :start 1) => 1
(count #\x "Some sequence") => 0
(count '(1 2) #(9 3 (1 2) 6 7 8)) => 0
(count '(1 2) #(9 3 (1 2) 6 7 8) :test #'equal) => 1
(count 1 #(0 1 0 0 0 1 0) :from-end t) => 2
```

elt

Syntax: Symbol type: function

elt *sequence index* => element

Argument description:

sequence	a sequence
index	valid sequence index

ELT function accesses specified elements of sequences. The index is counted from zero. Accessing out-of-bounds indices signals condition, or causes crash and/or undefined behavior, depending on compilation safety mode. Unlike AREF, ELT works on lists too.

ELT may by used with conjunction of SETF.

```
(elt "hola" 0) => #\h
(elt "hola" 3) => #\a
(elt #(5 3 6 8) 1) => 3
(elt '(5 3 6 8) 1) => 3

(let ((a (list 1 2 3 4))) (setf (elt a 1) 'x) a) => (1 X 3 4)
(let ((a (copy-seq "hola"))) (setf (elt a 1) #\0) a) => "h0la"
```

find

Syntax: Symbol type: function

find *item sequence test* (keyword) *from-end* (keyword) *start* (keyword) *end* (keyword) *key* (keyword) => element

Argument description:

item	an item to be found
sequence	a sequence to be searched
test	function key and item comparison
from-end	direction of search, default is NIL - forward
start	starting position for search, default is 0
end	final position for search, default is NIL - end of sequence
key	function for extracting value before test

FIND function searches for an element (item) satisfying the test. Return value is element itself or NIL if item is not found. See also POSITION, POSITION-IF, FIND, FIND-IF and MEMBER.

```
(find #\s "Some sequence") => #\s
(find #\s "Some sequence" :key #'char-downcase) => #\S
(find #\s "Some sequence" :key #'char-downcase :start 1) => #\s
(find #\x "Some sequence") => NIL
(find '(1 2) #(9 3 (1 2) 6 7 8)) => NIL
(find '(1 2) #(9 3 (1 2) 6 7 8) :test #'equal) => (1 2)
(find 1 #(0 1 0 0 0 1 0) :from-end t) => 1
```

length

Symbol type: function

length *seq* => integer

Argument description:

seq	sequence of objects

LENGTH function computes length of the list, vector, string or other sequences. For lists, LENGTH may get stuck in cyclic cons structures unlike LIST-LENGTH.

```
(length "hola") => 4
(length "") => 0
(length #(2 'a 5.6)) => 3
(length #*101010101110) => 12
(length (list 'a 'b 'c)) => 3
(length nil) => 0
(length '(a . (b . nil))) => 2
(length (cons "moo" nil)) => 1
(length (cons "moo" (cons "boo" nil))) => 2
```

make-array

Syntax: Symbol type: function

make-array *dimensions* *element-type* (keyword) *initial-element* (keyword) *initial-contents* (keyword) *adjustable* (keyword) *fill-pointer* (keyword) *displaced-to* (keyword) *displaced-index-offset* (keyword) => an array

Argument description:

dimensions	list of dimensions, or non-negative integer
element-type	a type specifier, default is T - any type
initial-element	a value, default is implementation dependent
initial-contents	an object
adjustable	a generalized boolean, default is NIL
fill-pointer	a valid fill pointer for the array, or T or NIL
displaced-to	an array or NIL, default is NIL
displaced-index-offset	a valid array row-major index for displaced arrays, default is 0

MAKE-ARRAY function creates a new array. Array can be adjustable if specified, that is its dimensions can be shrinked or enlarged by ADJUST-ARRAY function.

One-dimensional arrays can have a fill-pointer. Fill-pointer makes array look like as if it would be shorter with only as many elements as fill-pointer specifies - while elements at the real end of array a still retained. Such array can be very easily enlarged or shrinked in bounds of the real size just by setting fill-pointer which is very fast. Functions like VECTOR-PUSH, VECTOR-PUSH-EXTEND and VECTOR-POP make use of this.

Arrays can be displaced onto another array. Such array can have different dimensions and elements are shared on underlying row-major element order.

See also AREF, ELT, ADJUST-ARRAY, ARRAY-DIMENSION, ARRAY-DIMENSIONS, FILL-POINTER, ARRAY-IN-BOUNDS-P, ARRAY-ROW-MAJOR-INDEX, ARRAYP.

```
(make-array 5 :initial-element 'x) => #(X X X X X)
(make-array '(2 3) :initial-element 'x) => #2A((X X X) (X X X))
(length (make-array 10 :fill-pointer 4)) => 4
(array-dimensions (make-array 10 :fill-pointer 4)) => (10)
(make-array 10 :element-type 'bit :initial-element 0) => #*0000000000
(make-array 10 :element-type 'character :initial-element #\a) => "aaaaaaaaaa"
(let ((a (make-array '(2 2) :initial-element 'x :adjustable t))) (adjust-array a '(1 3) :initial-element 'y) a) => #2A((X X Y))
```

make-sequence

Syntax: Symbol type: function

make-sequence *result-type* *size* *initial-element* (keyword) => sequence

Argument description:

result-type	sequence type specifier
size	a non-negative integer

initial-element	element which is used to fill sequence, default is implementation dependent

MAKE-SEQUENCE creates a new sequence of specified type and number of elements. See also MAP.

```
(make-sequence 'list 4 :initial-element 'x) => (X X X X)
(make-sequence 'vector 4 :initial-element 'x) => #(X X X X)
(make-sequence 'vector 4 :initial-element #\a) => #(#\a #\a #\a #\a)
(make-sequence 'string 4 :initial-element #\a) => "aaaa"
```

map

Syntax: Symbol type: function

map *result-type fn seqs* (one or more) => sequence or NIL

Argument description:

result-type	sequence type specifier or NIL
fn	function that takes as many arguments as there are sequences
seqs	sequences which elements are processed in parallel

MAP applies function FN to elements of sequence with same index. Each application result is put into resulting sequence. Length of resulting sequence is the length of the shortest sequence in argument. Return value is NIL when NIL was specified as result-type. See also MAPC, MAPCAR and MAPCAN.

```
(map 'list (lambda (x) (+ x 10)) '(1 2 3 4)) => (11 12 13 14)

(map 'vector #'identity "hola") => #(#\h #\o #\l #\a)
(map '(vector character) #'identity #(#\h #\o #\l #\a)) => "hola"
(map 'string #'identity '(#\h #\o #\l #\a)) => "hola"

(map 'vector #'list '(123 symbol "string" 345) '(1 2 3)) => #((123 1) (SYMBOL 2) ("string" 3))

(map 'list #'* '(3 4 5) '(4 5 6)) => (12 20 30)
(map 'nil #'* '(3 4 5) '(4 5 6)) => NIL
```

map-into

Syntax: Symbol type: function

map-into *result-sequence fn seqs* (one or more) => result-sequence

Argument description:

result-sequence	sequence type specifier or NIL
fn	function that takes as many arguments as there are sequences
seqs	sequences which elements are processed in parallel

MAP-INTO applies function fn to elements of sequence with same index. Each application result is destructively put into resulting sequence. The iteration terminates when the shortest sequence (of any of the sequences or the result-sequence) is exhausted. Return value is same as the first argument. See also MAP, MAPCAR and MAPCAN.

```
(let ((a (list 1 2 3 4))) (map-into a #'* a a) a) => (1 4 9 16)
(let ((a (vector 1 2 3 4))) (map-into a #'* a a) a) => #(1 4 9 16)
(let ((a (vector 1 2 3 4))) (map-into a #'1+ '(1 2)) a) => #(2 3 3 4)
```

position

Syntax: Symbol type: function

position *item sequence test* (keyword) *from-end* (keyword) *start* (keyword) *end* (keyword) *key* (keyword) => index or NIL

Argument description:

item	an item to be found
sequence	a sequence to be searched
test	function key and item comparison
from-end	direction of search, default is NIL - forward
start	starting position for search, default is 0
end	final position for search, default is NIL - end of sequence
key	function for extracting value before test

POSITION function searches for an element (item) satisfying the test. Return value is index of such item or NIL if item is not found. Index is relative to start of the sequence regardless of arguments. See also POSITION-IF, FIND, FIND-IF and MEMBER.

```
(position #\s "Some sequence") => 5
(position #\s "Some sequence" :key #'char-downcase) => 0
(position #\s "Some sequence" :key #'char-downcase :start 1) => 5
(position #\x "Some sequence") => NIL
(position '(1 2) #(9 3 (1 2) 6 7 8)) => NIL
(position '(1 2) #(9 3 (1 2) 6 7 8) :test #'equal) => 2
(position 1 #(0 1 0 0 0 1 0) :from-end t) => 5
```

reduce

Syntax: Symbol type: function

reduce *fn seq initial-value* (keyword) *key* (keyword) *from-end* (keyword) *start* (keyword) *end* (keyword) => an object

Argument description:

fn	a two argument function
seq	a sequence
initial-value	an object
key	function for extracting values from sequence
from-end	direction flag, default is NIL
start	bounding index
end	bounding index

REDUCE applies function fn to its previous result and next element. The result is what fn returned in last call. For the first call fn is called with either initial-value and first element or first two elements. See also MAPCAR, MAPCAN, MAP.

```
(reduce #'list '(1 2 3 4)) => (((1 2) 3) 4)
(reduce #'list '(1 2 3 4) :initial-value 0) => ((((0 1) 2) 3) 4)
(reduce #'list '(1 2 3 4) :initial-value 0 :from-end t) => (1 (2 (3 (4 0))))
(reduce #'list '(1 2 3 4) :from-end t) => (1 (2 (3 4)))
(reduce (lambda (x y) (+ (* x 10) y)) '(1 2 3 4)) => 1234
(reduce #'+ '(1 2 3 4)) => 10
(reduce #'* '(1 2 3 4) :initial-value 1) => 24
```

remove

Syntax: Symbol type: function

remove *item seq from-end* (keyword) *test* (keyword) *test-not* (keyword) *start* (keyword) *end* (keyword) *count* (keyword) *key* (keyword) => sequence

Argument description:

item	an object
seq	a sequence
from-end	boolean specifying processing direction
test	equality test
test-not	non-equality test
start	bounding index, default 0

end	bounding index, default nil
count	integer for how many elements to remove, or nil
key	function of one argument

REMOVE make new sequence of the same type that has some elements removed. COUNT may limit the number of removed elements. See also REMOVE-IF, DELETE, DELETE-IF, SUBSEQ, and REMOVE-DUPLICATES.

```
(remove #\s "Sample string sequence") => "Sample tring equence"
(remove #\s "Sample string sequence" :count 1) => "Sample tring sequence"
(remove #\s "Sample string sequence" :test #'char-equal) => "ample tring equence"
(remove nil '(1 2 nil 4 nil 6)) => (1 2 4 6)
```

reverse

Syntax: Symbol type: function

reverse *seq* => a sequence

Argument description:

seq	a sequence

REVERSE function makes new sequence with reverted order of elements. See also MAP, MAPCAR and MAPCAN.

```
(reverse '(1 2 3 4)) => (4 3 2 1)
(reverse '#(1 2 3 4)) => #(4 3 2 1)
(reverse "hola") => "aloh"
```

search

Syntax: Symbol type: function

search *sequence1* *sequence2* *test* (keyword) *from-end* (keyword) *start1* (keyword) *start2* (keyword) *end1* (keyword) *end2* (keyword) *key* (keyword) => position

Argument description:

sequence1	a sequence to be found in sequence2
sequence2	a sequence to be searched
test	function key and item comparison
from-end	direction of search, default is NIL - forward
start1	starting position in sequence1, default is 0
start2	starting position in sequence2, default is 0
end1	final position in sequence1, default is NIL - end of sequence
end2	final position in sequence2, default is NIL - end of sequence
key	function for extracting value before test

SEARCH function searches for one sequence in another. See also POSITION, POSITION-IF, FIND, FIND-IF and MEMBER.

```
(search "lo" "hello world") => 3
(search "lo" "HelLo WoRLd" :key #'char-upcase) => 3
(search "lo" "HelLo WoRLd") => NIL
```

some

Syntax: Symbol type: function

some *predicate* *sequences* (one or more) => T or NIL

Argument description:

predicate	predicate function
sequences	sequences

SOME function searches the sequences for values for which predicate returns true. It there is such list of values that occupy same index in each sequence, return value is true, otherwise false.

```
(some #'alphanumericp "") => NIL
```

```
(some #'alphanumericp "...") => NIL
(some #'alphanumericp "ab...") => T
(some #'alphanumericp "abc") => T

(some #'< '(1 2 3 4) '(2 3 4 5)) => T
(some #'< '(1 2 3 4) '(1 3 4 5)) => T
(some #'< '(1 2 3 4) '(1 2 3 4)) => NIL
```

string

Syntax: Symbol type: function

string *object* => string

Argument description:

object an object

STRING function converts symbols, characters and possibly some other types into a string. If object is of string type, it is directly returned.

```
(string 'moo) => "MOO"
(string #\a) => "a"
(string "some string") => "some string"
```

string-downcase

Syntax: Symbol type: function

string-downcase *string* *start* (keyword) *end* (keyword) => string

Argument description:

string a string
start integer bouded by string length
end integer bouded by string length

STRING-DOWNCASE function converts string into its upcase reprezentation. returned. See also STRING-UPCASE, STRING-CAPITALIZE, CHAR-UPCASE and CHAR-DOWNCASE.

```
(string-downcase "SOME STRING") => "some string"
(string-downcase "SOME STRING" :start 2) => "SOme string"
(string-downcase "SOME STRING" :start 2 :end 8) => "SOme strING"
```

string-upcase

Syntax: Symbol type: function

string-upcase *string* *start* (keyword) *end* (keyword) => string

Argument description:

string a string
start integer bouded by string length
end integer bouded by string length

STRING-UPCASE function converts string into its upcase reprezentation. returned. See also STRING-DOWNCASE, STRING-CAPITALIZE, CHAR-UPCASE and CHAR-DOWNCASE.

```
(string-upcase "some string") => "SOME STRING"
(string-upcase "some string" :start 2) => "soME STRING"
(string-upcase "some string" :start 2 :end 8) => "soME STRing"
```

subseq

Syntax: Symbol type: function

subseq *seq* *start* *end* (optional) => sequence

Argument description:

seq a sequence
start bounding index
end bounding index, default NIL

SUBSEQ function makes new sequence as a subseqence of argument. Default ending index is end of

sequence. See also COPY-SEQ and MAP.

SUBSEQ may be used with SETF.

```
(subseq "hello world" 3) => "lo world"
(subseq "hello world" 3 5) => "lo"
(let ((a "hello world")) (setf (subseq a 3 5) "LO") a) => "helLO world"
(let ((a "hello world")) (setf (subseq a 3 5) "YYY") a) => "helYY world"
```

vector

Syntax: Symbol type: function

vector *list* (zero or more) => vector

Argument description:

list list of objects

VECTOR function makes new simple general vector from arguments. See also LIST.

```
(vector 1 2 3) => #(1 2 3)
(vector 'a #c(1 2) "moo") => #(A #C(1 2) "moo")
(elt (vector 1 2 3) 0) => 1
(elt (vector 1 2 3) 1) => 2
(vector) => #()
(equal #(1 2 3) (vector 1 2 3)) => NIL
(equalp #(1 2 3) (vector 1 2 3)) => T
(type-of (vector 1 2 3)) => (SIMPLE-VECTOR 3)
```

vector-pop

Syntax: Symbol type: function

vector-pop *vector* => an object

Argument description:

vector a vector with fill pointer

VECTOR-POP function pops a element from specified vector. Supplied vector must have fill-pointer (see MAKE-ARRAY). Fill-pointer is decremented. The element to be popped is found at new fill-pointer position. See also MAKE-ARRAY, VECTOR-POP and VECTOR-PUSH. Return value is object found at previous end of vector.

```
(defparameter *v* (make-array 2 :fill-pointer 0)) => *V*
(vector-push 4 *v*) => 0
(vector-push 3 *v*) => 1
*v* => #(4 3)
(vector-pop *v*) => 3
(vector-pop *v*) => 4
```

vector-push

Syntax: Symbol type: function

vector-push *new-element* *vector* => index or NIL

Argument description:

new-element a object

vector a vector with fill pointer

VECTOR-PUSH function pushes new-element into specified vector. Supplied vector must have fill-pointer (see MAKE-ARRAY). New element is placed at last fill-pointer position and fill-pointer is incremented. See also MAKE-ARRAY, VECTOR-POP and VECTOR-PUSH. Return value is index at which the new item was placed, or NIL if there is no room.

```
(defparameter *v* (make-array 2 :fill-pointer 0)) => *V*
(vector-push 4 *v*) => 0
(vector-push 3 *v*) => 1
(vector-push 2 *v*) => NIL
*v* => #(4 3)
(vector-pop *v*) => 3
(vector-pop *v*) => 4
```

vector-push-extend

Syntax: Symbol type: function

vector-push-extend *new-element* *vector* *extension* (keyword) => index

Argument description:

new-element	a object
vector	a vector with fill pointer
extension	a positive integer

VECTOR-PUSH-EXTEND function pushes new-element into specified vector. Supplied vector must be adjustable (see MAKE-ARRAY) and have fill-pointer. New element is placed at last fill-pointer position and fill-pointer is incremented. Vector size is adjusted a number of items as specified by extension argument, if necessary. See also MAKE-ARRAY, VECTOR-POP and VECTOR-PUSH. Return value is index at which the new item was placed.

```
(defparameter *v* (make-array 2 :fill-pointer 0 :adjustable t)) => *V*
(vector-push-extend 4 *v*) => 0
(vector-push-extend 3 *v*) => 1
(vector-push-extend 2 *v*) => 2
*v* => #(4 3 2)
(vector-pop *v*) => 2
(vector-pop *v*) => 3
(vector-pop *v*) => 4
```

Symbol, Characters, Hash, Structure, Objects and Conversions

atom

Syntax: Symbol type: function

atom *object* => T or NIL

Argument description:

object	an object

ATOM function returns true if the argument is not a cons cell, otherwise it returns false. See CONS and LIST.

```
(atom nil) => T
(atom 'some-symbol) => T
(atom 3) => T
(atom "moo") => T
(atom (cons 1 2)) => NIL
(atom '(1 . 2)) => NIL
(atom '(1 2 3 4)) => NIL
(atom (list 1 2 3 4)) => NIL
```

coerce

Syntax: Symbol type: function

coerce *object* *result-type* => an object

Argument description:

object	an object
result-type	a type specifier

COERCE function converts between different types. See full documentation for conversion description.

```
(coerce '(a b c) 'vector) => #(A B C)
(coerce #(a b c) 'list) => (A B C)
(coerce 4.4d0 'single-float) => 4.4
(coerce 4.4s0 'double-float) => 4.400000095367432d0
(coerce "x" 'character) => #\x
```

gethash

Syntax: Symbol type: function

gethash *key* *hashtable* *default* (optional) => an object

Argument description:

key	an object
hashtable	a hash-table
default	an object, default is NIL

GETHASH function reads associated value for given key in hashtable. (SETF GETHASH) adds or replaces associated values. See also MAKE-HASH-TABLE.

```
(defparameter *tab* (make-hash-table)) => *TAB*
(gethash 'x *tab*) => NIL, NIL
(setf (gethash 'x *tab*) "x") => "x"
(setf (gethash 'y *tab*) "yy") => "yy"
(gethash 'x *tab*) => "x", T
(gethash 'y *tab*) => "yy", T
(gethash 'z *tab* 'moo) => MOO, NIL
```

intern

Syntax: Symbol type: function

intern *stirng* *package* (optional) => symbol, status

Argument description:

stirng	a string
package	a package designator, default is current package

INTERN function makes a new symbol from string. Possible status values are: :inherited, :external, :internal, or nil.

```
(intern "MOO") => MOO, NIL
(intern "MOO") => MOO, :INTERNAL
(intern "moo") => |moo|, NIL
```

make-hash-table

Syntax: Symbol type: function

make-hash-table *test* (keyword) *size* (keyword) *rehash-size* (keyword) *rehash-threshold* (keyword) => hash-table

Argument description:

test	EQ, EQL EQUAL or EQUALP; default is EQL
size	a non-negative integer
rehash-size	a real number
rehash-threshold	a real number

MAKE-HASH-TABLE creates a new hash-table. Size parameter specifies initial size of inner table. Test specifies comparison operator for keys. See also GETHASH.

```
(defparameter *tab* (make-hash-table)) => *TAB*
(gethash 'x *tab*) => NIL, NIL
(setf (gethash 'x *tab*) "x") => "x"
(setf (gethash 'y *tab*) "yy") => "yy"
(gethash 'x *tab*) => "x", T
(gethash 'y *tab*) => "yy", T
(gethash 'z *tab* 'moo) => MOO, NIL
```

Input and output

format

Syntax: Symbol type: function

format *destination* *control-string* *args* (zero or more) => string or NIL

Argument description:

destination	T, NIL, stream or string with fill-pointer
control-string	a string with formating directives
args	format arguments for control-string

FORMAT function does a complex text formatting. Formatting rules are driven by control-string and

arguments in arg. When destination is stream or string with fill-pointer, the resulting string is written to it. T as a destination means "write to terminal". NIL as destination means "return the formatted string back as string". See also WRITE-STRING, TERPRI, PRINC, PRIN1 and PRINT.

Control string is composed of normal text and embedded directives. Directives begin with tilde (~) character. Most common are: ~a - output with aesthetics, ~s - standard output, ~% newline, tilde parenthesis - flow control, tilde tilde - escape sequence for tilde. See full documentation or examples for more.

```
(format nil "Items in list:~%~{~a, ~}" '(1 2 3 4)) => "Items in list:
1, 2, 3, 4, "
(format nil "~{~a~^, ~}" '(1 2 3 4)) => "1, 2, 3, 4"
(format nil "~f" 3.141592) => "3.141592"
(format nil "~2,3f" 3.141592) => "3.142"
(format nil "~7,3f" 3.141592) => "  3.142"
(format nil "~a ~s" "xyz" "xyz") => "xyz \"xyz\""
```

read

Syntax: Symbol type: function

read *input-stream* (optional) *eof-error-p* (optional) *eof-value* (optional) *recursive-p* (optional) => an object

Argument description:

input-stream	an input stream, default is standard input
eof-error-p	a boolean, true (default) is EOF should be signaled
eof-value	an object that is returned as EOF value
recursive-p	flag to note recursive processing

READ function reads arbitrary readable lisp object from input stream. Reading process uses *read-table*. Note that *read-eval* global variable controls read-time evaluation (#. macro).

```
(let ((s (make-string-input-stream "(1 2 3)"))) (read s)) => (1 2 3)
(let ((s (make-string-input-stream "#(1 2 3)"))) (read s)) => #(1 2 3)
(let ((s (make-string-input-stream "\"hola\""))) (read s)) => "hola"
```

read-char

Syntax: Symbol type: function

read-char *input-stream* (optional) *eof-error-p* (optional) *eof-value* (optional) *recursive-p* (optional) => char

Argument description:

input-stream	an input stream, default is standard input
eof-error-p	a boolean, true (default) is EOF should be signaled
eof-value	an object that is returned as EOF value
recursive-p	flag to note recursive processing

READ-CHAR function reads a character from input stream.

```
(let ((s (make-string-input-stream (format nil "line 1~%line 2~%line 3")))) (read-char s))
=> #\l
```

read-line

Syntax: Symbol type: function

read-line *input-stream* (optional) *eof-error-p* (optional) *eof-value* (optional) *recursive-p* (optional) => line, missing-newline-p

Argument description:

input-stream	an input stream, default is standard input
eof-error-p	a boolean, true (default) is EOF should be signaled
eof-value	an object that is returned as EOF value
recursive-p	flag to note recursive processing

READ-LINE function reads a line from input stream into string.

```
(let ((s (make-string-input-stream (format nil "line 1~%line 2~%line 3")))) (read-line s))
=> "line 1", NIL
```

write-string

Syntax: Symbol type: function

write-string *string* *output-stream* (optional) *start* (keyword) *end* (keyword) => string

Argument description:

string	a string
output-stream	a stream, default is standard output
start	bounding index
end	bounding index

WRITE-STRING function writes string into standard output or specified output stream. See WRITE-LINE, FORMAT.

```
(write-string "xyz")
xyz
=> "xyz"
```

Functions, Evaluation, Flow Control, Definitions and Syntax

apply

Syntax: Symbol type: function

apply *fn* *args* (zero or more) => value

Argument description:

fn	a function designator
args	call arguments

APPLY function call supplied function with specified arguments. Argument list is constructed as (append (butlast args) (first (last args))). Note that there is limitation of maximal number of arguments, see CALL-ARGUMENTS-LIMIT constant. See also FUNCALL, LAMBDA.

```
(apply #'+ 1 2 3 '(4 5 6)) => 21
(apply #'sin '(1.0)) => 0.84147096
(apply #'sin 1.0 nil) => 0.84147096
```

case

Syntax: Symbol type: macro

case *expression* *variants* (zero or more) => an object

Argument description:

expression	a value used to distinguish between variants
variants	list of match-values and code variants

CASE macro is used for branching. Variants are tested sequentially for EQL equality with from the top. See also IF, CASE.

```
(case (+ 1 2)
      (5 "variant 1, five")
      ((2 3) "variant 2, two or three")
      (otherwise "variant 3, none of above")) => "variant 2, two or three"
```

cond

Syntax: Symbol type: macro

cond *variants* => an object

Argument description:

variants	list of test and code variants

COND macro is used for branching. Variants are tested sequentially from the top. See also IF, CASE.

```
(cond ((> 3 4) "variant 1")
      ((> 4 2) "variant 2")
      (t "always valid variant")) => "variant 2"
```

defparameter

Syntax: Symbol type: macro

defparameter *name* *initial-value* *documentation* (optional) => an object

Argument description:

name	a name for global variable
initial-value	an expression
documentation	documentation string

DEFPARAMETER defines global variable with dynamic scoping. Usual conventions dictate to make such variables easy to distinguish so their name is surrounded by stars. Value for variable is reevaluated for each occurence (unlike with DEFVAR). See also DEFVAR, LET, SETQ.

```
(defparameter *my-global-variable* (+ 3 5)) => *MY-GLOBAL-VARIABLE*
*my-global-variable* => 8
```

defun

Syntax: Symbol type: macro

defun *name* *args* *forms* (zero or more) => symbol

Argument description:

name	symbol
args	arguments of function
forms	sequentially executed forms

DEFUN form creates named function. The function is associated with definition environment. Named functions can be called simply by specifying their name in function position in parenthesis, or they can be acquired by FUNCTION special form, or SYMBOL-FUNCTION function. Arguments of function can be regular (matched by position), optional (with default values), keyword (matched by keyword symbol) and rest (taking rest of arguments into a list). Result of function application is value of the last form unless return function or nonlocal exit is executed. Functions can be redefined. See also LAMBDA, FUNCALL, APPLY.

```
(defun myname (x) (+ x 3)) => MYNAME

(defun myname (x y) (* x y) (+ x y)) (myname 2 3) => 5

(defun myname (&optional (one 1) (two 2)) (list one two)) (myname) => (1 2)
(defun myname (&optional (one 1) (two 2)) (list one two)) (myname 10) => (10 2)
(defun myname (&optional (one 1) (two 2)) (list one two)) (myname 10 20) => (10 20)

(defun myname (&rest myargs) (length myargs)) (myname) => 0
(defun myname (&rest myargs) (length myargs)) (myname 4 5 6) => 3
(defun myname (&rest myargs) (length myargs)) (myname '(4 5 6)) => 1

(defun myname (&key one two) (list one two)) (myname) => (NIL NIL)
(defun myname (&key one two) (list one two)) (myname :two 7) => (NIL 7)
(defun myname (&key one two) (list one two)) (myname :two 7 :one 4) => (4 7)
```

eval

Syntax: Symbol type: function

eval *form* => value

Argument description:

form	a value forming lisp expression

EVAL function interprets (or compiles and runs) the argument and returns the result. See also APPLY, LAMBDA, FUNCALL.

```
(eval '(+ 1 2)) => 3
(eval '(let ((x 2)) (sin x))) => 0.9092974
(let ((expr '((x 2)) (sin x)))) (eval (cons 'let expr))) => 0.9092974
```

flet

Syntax: Symbol type: special form

flet *bindings* *body* (zero or more) => an object

Argument description:

bindings	list containing function definitions
body	program code in which definitions above are effective, implicit progn

FLET is special form for local function binding. Bindings are not recursive and cannot refer to each other. Each binding contains function name, arguments, and function body. See LABELS, DEFUN, LAMBDA.

```
(flet ((sin2x (x) (sin (* 2 x)))
       (cos2x (x) (cos (* 2 x))))
 (+ (sin2x 0.2) (cos2x 0.2)))
=> 1.3104793
```

funcall

Syntax: Symbol type: function

funcall *fn* *args* (zero or more) => value

Argument description:

fn	a function designator
args	call arguments

FUNCALL function call supplied function with specified arguments. Argument list is same as in the rest of funcall call. Function designator is function itself or symbol specifying global function name. Note that there is limitation of maximal number of arguments, see CALL-ARGUMENTS-LIMIT constant. See also APPLY, LAMBDA.

```
(funcall #'+ 1 2 3 4 5 6) => 21
(funcall #'sin 1.0) => 0.84147096
(funcall 'sin 1.0) => 0.84147096
```

function

Syntax: Symbol type: special form

function *symbol* => function

Argument description:

symbol	symbol of function name

FUNCTION is special form for accessing namespace of functions. See also QUOTE.

```
(function sin) => #<FUNCTION>
#'sin => #<FUNCTION>
(funcall #'sin 1.0) => 0.84147096
```

if

Syntax: Symbol type: special syntax

if *test* *then* *else* (optional) => an object

Argument description:

test	an expression
then	an expression
else	an expression, default NIL

IF special form is used for branching. Either "then" or "else" branch is taken. Then branch is selected when "test" result is not NIL. See also COND, CASE.

```
(if (> 3 4) "variant 1" "variant 2") => "variant 2"
(if (> 4 3) "variant 1" "variant 2") => "variant 1"
```

labels

Symbol type: special form

labels *bindings* *body* (zero or more) => an object

Argument description:

bindings	list containing function definitions
body	program code in which definitions above are effective, implicit progn

LABELS is special form for local function binding. Bindings can be recursive and can refer to each other. Each binding contains function name, arguments, and function body. See FLET, DEFUN, LAMBDA.

```
(labels ((fact2x (x) (fact (* 2 x)))
         (fact (x) (if (< x 2) 1 (* x (fact (1- x))))))
  (fact2x 3))
 => 720
```

lambda

Syntax: Symbol type: special

lambda *args* *forms* (zero or more) => function

Argument description:

args	arguments of function
forms	sequentially executed forms

LAMBDA form creates function object associated with definition environment. This function object is called "closure". It can be applied later with funcall. Arguments of function can be regular (matched by position), optional (with default values), keyword (matched by keyword symbol) and rest (taking rest of arguments into a list). Lambda form don't have to be prefixed with "#'" syntax. Result of function application is value of the last form unless return function or nonlocal exit is executed.

```
(lambda (x) (+ x 3)) => <#closure>

(funcall (lambda (x y) (* x y) (+ x y)) 2 3) => 5

(funcall (lambda (&optional (one 1) (two 2)) (list one two))) => (1 2)
(funcall (lambda (&optional (one 1) (two 2)) (list one two)) 10) => (10 2)
(funcall (lambda (&optional (one 1) (two 2)) (list one two)) 10 20) => (10 20)

(funcall (lambda (&rest myargs) (length myargs))) => 0
(funcall (lambda (&rest myargs) (length myargs)) 4 5 6) => 3
(funcall (lambda (&rest myargs) (length myargs)) '(4 5 6)) => 1

(funcall (lambda (&key one two) (list one two))) => (NIL NIL)
(funcall (lambda (&key one two) (list one two)) :two 7) => (NIL 7)
(funcall (lambda (&key one two) (list one two)) :two 7 :one 4) => (4 7)
```

let

Syntax: Symbol type: special form

let *bindings* *body* (zero or more) => an object

Argument description:

bindings	list of variable - initial value pairs
body	program code in which definitions above are effective, implicit progn

LET is special form for variable binding. Bindings are described in two element lists where the first element specifies name and the second is code to compute its value, or single variable without default initialization. There are also declarations possible before body.

```
(let (a b (c 3) (d (+ 1 2))) (list a b c d)) => (NIL NIL 3 3)
```

progn

Syntax: Symbol type: special form

progn *list* => value

Argument description:

list	expressions

PROGN calls its expression in the order they have been written. Resulting value is the value of the last form unless non-local control flow forced earlier return. See also PROG1, PROG2.

Note that many macros and special forms behave partially as PROGN. It is called "implicit progn".

```
(progn 1 2 3 4 5) => 5
(progn 1 2 (sin 2.0) 4 (sin 1.0)) => 0.84147096
(progn) => NIL
```

quote

Syntax: Symbol type: special form

quote *data* => value

Argument description:

data data

QUOTE is special form for data quotation. The apostrophe character is reader macro synonym for QUOTE. See also FUNCTION.

```
(+ 1 2 3) => 6
(quote (+ 1 2 3)) => (+ 1 2 3)
'(+ 1 2 3) => (+ 1 2 3)
(let ((some-symbol 4)) some-symbol) => 4
(let ((some-symbol 4)) (quote some-symbol)) => SOME-SYMBOL
(let ((some-symbol 4)) 'some-symbol) => SOME-SYMBOL
```

setf

Syntax: Symbol type: macro

setf *pairs* (zero or more) => an object

Argument description:

pairs pairs of places and values

SETF is similar to SETQ but works with generalized places. Many functions for read access can be turned into write access. See LET, SETQ. SETF expanders can be defined in multiple ways, most easier is (defun (setf my-name) arguments body...).

```
(let (a b) (setf a 4) (setf b 3) (setf a (+ a b))) => 7
(let ((a #(1 2 3 4))) (setf (aref a 2) 'new-value) a) => #(1 2 NEW-VALUE 4)
(let ((a '(1 2 3 4))) (setf (third a) 'new-value) a) => (1 2 NEW-VALUE 4)
```

setq

Syntax: Symbol type: special form

setq *pairs* (zero or more) => an object

Argument description:

pairs pairs of variables and values

SETQ special form sets a variable. If a variable name is not know, implementation may create new one global and dynamic one. See also LET, SETF.

```
(let (a b) (setq a 4) (setq b 3) (setq a (+ a b))) => 7
```