LISP Introduction

This document is taken from the tutorials of PWGL
— a music software based on LispWorks, now deprecated —
re-edited for PDF format by Julien Vincenot.

Table of Contents :

Quick introduction to Lisp

PWGL is based on Lisp. Learning Lisp is highly recommended and beneficial as it allows :

- to better understand the visual Patch,
- to use PWConstraints and ENP-Script,
- to understand the various Lisp-based representations (such as ENP-score-notation),
- to create functions, and
- to create user-libraries.

Lisp is a family of computer programming languages with a long history and a distinctive fully-parenthesized syntax. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older. Like Fortran, Lisp has changed a great deal since its early days, and a number of dialects have existed over its history. Today, the most widely-known general-purpose Lisp dialects are Common Lisp and Scheme. (Wikipedia)

For further information about Lisp see :
http://en.wikipedia.org/wiki/Lisp_(programming_language)

Here are some recommended and free Online books about Lisp :
(1) *Common Lisp: A Gentle Introduction to Symbolic Computation*
http://www.cs.cmu.edu/~dst/LispBook/index.html

(2) *Practical Common Lisp*
http://www.gigamonkeys.com/book/

(3) *Common Lisp the Language, 2nd Edition*
http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/clm/node1.html

If you are interested in learning Lisp outside PWGL environment, a good alternative is LispWorks Common Lisp. PWGL itself is programmed with LispWorks and a free Personal Edition can be downloaded from the companys web site :
http://www.lispworks.com/downloads/index.html

I. Syntax : Functions vs. Data

The syntax of Lisp is simple. Lisp expressions are written inside parentheses and the elements are separated by spaces.

Functions

The function operates on the data and returns a result. In a list the first element specifies the function. The remaining elements are the arguments, e.g :

```
> (+ 1 2 3)
6

> (1- (* (/ 5 6) pi))
1.6179938779914945D0

> (defun sub-one(x) (- x 1))
SUB-ONE
```

Data

The term data means information, such as numbers, words, or lists of things. To make a list data you need to use a special form called QUOTE. QUOTE returns its argument without evaluating it.

Without the QUOTE, Lisp interpretes expressions as function calls (as described above). The special form quote tells Lisp not to evaluate the form but to treat it as a literal (i.e., data). There is a shorthand for QUOTE function that is used by Lisp programmers almost without exceptions and that is an apostrophe ('), e.g., '(a b c d e) is the same as (quote (a b c d e)). The following demonstrates the difference between quoted and not quoted expressions :

```
                > '(a b c d e)
                (a b c d e)

                > (a b c d e)

        Error: Undefined operator A in form (A B C D E).
        1 (continue) Try invoking A again.
        2 Return some values from the form (A B C D E).
        3 Try invoking SYSTEM::A with the same arguments.
        4 Set the symbol-function of A to the symbol-function of SYSTEM::A.
        5 Try invoking something other than A with the same arguments.
        6 Set the symbol-function of A to another function.
        7 Set the macro-function of A to another function.
        8 (abort) Return to level 0.
        9 Return to top loop level 0.

        Type :b for backtrace, :c <option number> to proceed,
        or :? for other options
```

IMPORTANT :

It is not necessary to quote so called self-evaluating objects such as numbers, T or NIL !

II. Symbols

A symbol is a sequence of letters, digits, and permissible special characters (e.g., _,-,*,?,!). Numbers are not symbols. The characters colon (:), semi-colon (;) and comma (,) are to be avoided because they have special meaning in Lisp. Symbols are names that are used to bind to a value.

Symbols can be used, for example, to :

- store values,
- recall values,
- bind temporary values, and
- use as arguments to functions.

Here is a list of some valid symbols :

- midis
- note1
- 24overtones
- all-card-3-pcs
- *current-harmony*

There are two special symbols, T and NIL. Lisp uses T and NIL to represent true and false. Symbols like T and NIL are called 'self-evaluating symbols', because they evaluate to themselves.

Symbols that starts with a colon are keywords :

```
> :notes
:NOTES

> :pitches
:PITCHES
```

A symbol evaluates to the value of the variable it refers to. To access the symbol itself, it must be quoted.

III. Numbers

There are three kinds of numbers that are used most commonly, integers, floating point numbers, and ratios.

Integers

Integers are whole numbers, e.g., 1, 12, 18446744073709551616.

Floating point numbers

A floating point number is always written with a decimal point,
e.g., 5.0, 3.141592653589793.

Ratios

Ratios are numbers consisting of nominator and denominator,
e.g., 1/2, 2/3, 3/5, 5/8, 8/13.
Common Lisp automatically simplifies ratios to use the smallest possible denominator,
e.g., the ratios 4/6, 6/9, and 10/15 are all simplified to 2/3.

Arithmetic Functions

The standard arithmetic functions are available: +, -, *, /, FLOOR, CEILING, MOD, SIN, COS, TAN, SQRT, EXP, EXPT, etc. All of them accept any kind of number as an argument :

```
> (+ 6 1/2)
13/2
> (mod 23 12)
11
> (+ 4 8)
12
```

IV. Lists

List is the most important data type in Lisp. Lists can be used to represent practically anything : sets, melodies, rhythms, etc.

Lists consist of items enclosed in parentheses. These items are called the elements of the list. Here are some valid lists :

- (5 4 0 9 7 2 8 1 3 6 10 11)
- ((0 4 7) (0 5 9) (2 6 9) (2 7 11))
- ((:motive1 (0 1 3 2)) (:motive2 (0 6 2)))
- ((1 (1 1 1)) (1 (1.0 1)) (1 (1.0 1 1 1)) (2 (1)))

Creating Lists

There are several ways to create lists :

- using quote, e.g., '(0 1 2 3 4), '(a b c d);
- using the LIST function, e.g. (list 0 1 2 3 4), (list 'a 'b 'c 'd);
- using iteration, e.g., (iter (for x from 1 to 5) (collect x)).

```
> '(0 1 2 3 4)
(0 1 2 3 4)

> (list 0 1 2 3 4)
(0 1 2 3 4)

> (iter (for x from 0 to 4) (collect x))
(0 1 2 3 4)
```

V. Binding

Creating a new variable and giving it a value is called binding. If the variable appears inside LET or LET* form, the variable is locally bound. If the variable is defined using DEFPARAMETER or DEFVAR, it is globally bound. Locally bound variables can be accessed only inside the form they were created. Globally bound variables can be accessed and modified anywhere.

Global Binding

```
> (defparameter *harmonic-spectrum-on-110Hz*
        '(45 57 64 69 73 76 79 81 83 85))
```

Notice the use of asterisks (*) around the name of a global variable. This habit is highly recommended to distinguish between locally and globally bound variables.

Local Binding

LET binds base-note to 60, transposition-interval to 6, and executes the statements in its

```
> (let ((base-note 60)
     (transposition-interval 6))
  (+ base-note transposition-interval))
66
```

body.

The LET* special form is just like let except that it allows values to reference variables defined earlier in the same let* form, e.g.

```
> (let* ((freq1 440.0)
      (freq2 (* freq1 2)))
  freq2)
880.0
```

VI. Booleans and Conditionals

Lisp also provides a set of special forms for conditional execution, e.g., IF, WHEN, and UNLESS.

IF

The first argument of if is the condition that determines if the second or third argument will be executed :

```
> (if T 1 2)
1

> (if NIL 1 2)
2

> (if 1 2 3)
2
```

To put more than one statement in the then or else clause you need use the PROGN special form. PROGN executes each statement in its body and returns the value of the final one.

```
> (let ((a 6)
        (b 7)
        (c 8))
      (if (> a 5)
          (progn
            (setq a (+ b 7))
            (setq b (+ c 8)))
          (setq b 4)))
16
```

WHEN and UNLESS

An IF statement that does not have either a then or an else clause can be written using WHEN or UNLESS :

```
> (when T 3)
3

> (when NIL 3)
NIL

> (unless T 3)
NIL

> (unless NIL 3)
3
```

WHEN and UNLESS allow any number of statements in their bodies.

```
> (when t
   (setq a 5)
   (+ a 6))
11
```

COND and CASE

More complicated conditionals can be defined using COND, which is equivalent to an IF... ELSE IF. The COND function takes any arbitrary number of arguments. The arguments of COND consist of a condition and a result. The COND finds the first argument whose condition evaluates to true and then executes the corresponding action and returns the resulting value. None of the remaining conditions are evaluated :

```
> (let ((midi 61))
      (cond
        ((< midi 60) 'low-range)
        ((> midi 80) 'high-range)
        (T 'mid-range)))
MID-RANGE
```

The Lisp CASE statement allows the conditional execution of one of the forms by matching a test key :

```
> (let ((x :es))
     (case x
       ((:es :as :bb) 'flat)
       ((:cis :fis) 'sharp)
       (T 'natural)))
FLAT
```

The last T clause means that if x is not any of the keywords the CASE statement will return NATURAL.

VII. Iteration

ITERATE

ITERATE is an iteration construct for Common Lisp. It is similar to the LOOP macro provided by default in Common Lisp. It has, among other things, the following characteristics :

- it is extensible,
- it has a more Lisp-like syntax than LOOP,
- it isn't part of the ANSI standard for Common Lisp.

The ITERATE manual can be found here :
http://common-lisp.net/project/iterate/doc/index.html

```
> (iter (for midi from 60 to 84) (collect midi))
(60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84)


> (iter (for pc from 0 to 11 by 2) (collect pc))
(0 2 4 6 8 10)


> (iter (for freq first 60.0 then (* freq 2)) (repeat 8)
(collect freq))
(60.0 120.0 240.0 480.0 960.0 1920.0 3840.0 7680.0)


> (let ((midis '(60 64 65 63 64 68)))
  (iter (for midi1 in midis)
   (for midi2 in (cdr midis))
   (collect (- midi2 midi1))))
(4 1 -2 1 4)


> (iter (for x in '(60 61 62 63))
  (appending
   (iter (for y in '(60 61 62 63))
        (collect (list x y)))))
((60 60) (60 61) (60 62) (60 63) (61 60) (61 61)
 (61 62) (61 63) (62 60) (62 61) (62 62) (62 63)
 (63 60) (63 61) (63 62) (63 63))
```

DOLIST

DOLIST binds a variable to the elements of a list in order and stops in the end of the list.

```
> (dolist (x '(60 61 62 63))
(print x))
60
61
62
63
NIL
```

IMPORTANT :
DOLIST always returns NIL. Note that the value of x in the above example was never NIL : the NIL below the 63 was the value that DOLIST returns.

DOTIMES

DOTIMES is like DOLIST except that it iterates over integers.

```
> (dotimes (i 4) (print (* i i)))
0
1
4
9
16
NIL
```

VIII. FUNCALL, APPLY, and MAPCAR

FUNCALL

FUNCALL calls its first argument on its remaining arguments.

```
> (funcall #'+ 60 1)
61
```

APPLY

APPLY is like FUNCALL, except that its final argument should be a list. The elements of the list are treated as if they were additional arguments to a funcall.

MAPCAR

```
> (apply #'+ '(60 1))
61
```

The first argument to MAPCAR must be a function of one argument. MAPCAR applies this function to each element of a list and collects the results in another list.

```
> (mapcar #'(lambda(x) (eql x :rest))
      '(:note :rest :note :rest :note :rest))
(NIL T NIL T NIL T)
```

IX. Lambda

A LAMBDA function is a temporary nameless function. It can be used for several different purposes :

```
> (funcall #'(lambda (x) (+ x 6)) 60)
66

> (mapcar #'(lambda (x) (+ x 2)) '(60 61 62 63))
(62 63 64 65)
```

X. Sorting

Lisp provides two primitives for sorting : SORT and STABLE-SORT. The first argument to SORT is a list, the second is a comparison function. The STABLE-SORT function is exactly like sort, except that it preserves the original order of identical elements :

```
> (sort '(60 64 56 45 66 87) #'<)
(45 56 60 64 66 87)

> (sort '(60 64 56 45 66 87) #'>)
(87 66 64 60 56 45)
```

IMPORTANT :
SORT is allowed to destroy its argument. If the original sequence is important, you need to make a copy of it using the COPY-LIST or COPY-SEQ function.

XI. Equality

Numerical equality in Lisp is denoted by =. Two symbols are EQ if and only if they are identical. Two copies of the same list are not EQ, but they are EQUAL. If you want to be sure you can almost always use EQUAL. Only things that EQUAL is not able to tell equal are strings with different capitalization (e.g.,Flute" andflute" are not EQUAL) and objects.

```
> (eq 'fis 'fis)
T

> (eq 'fis 'cis)
NIL

> (= 60 60)
T

> (eq '(60 61 62) '(60 61 62))
NIL

> (equal '(60 61 62) '(60 61 62))
T

> (eql 'cis 'cis)
T

> (eql 60 60)
T
```

XII. DEFUN

DEFUN allows to define new Lisp functions.

```
> (defun pc? (midi)
   (mod midi 12))
PC?

> (pc? 24)
0



> (defun intervals(midis)
   (mapcar #'- (cdr midis) midis))
INTERVALS

> (intervals '(60 65 64 62))
(5 -1 -2)
```

Lisp Reference

A list of some useful Lisp functions, macros and special forms.

+
args: &rest numbers
[Function / Variable] as a function, adds all the arguments and returns the sum.

```
> (+ 5 6)
11

> (+ 5 6 7 8)
26
```

1+
args: number
[Function] returns the result of adding 1 to number.

```
> (1+ 5)
6
```

–
args: number &rest more-numbers
[Function / Variable] as a function subtracts each argument, from left to right, from the result of the previous subtraction, and returns the final difference.

```
> (– 5 6)
–1
> (– 5 6 7 8)
–16
```

1–
args: number
[Function] returns the result of subtracting 1 from number.

```
> (1– 5)
4
```

*

args: &rest numbers

[Function / Variable] as a function, multiplies all the numbers, returning the product.

```
> (* 5 6)
30
> (* 5 6 7 8)
1680
```

/

args: number &rest more-numbers

[Function / Variable] as a function divides each argument, from left to right, into the result of the previous division, and returns the final quotient.

```
> (/ 50 6)
25/3
> (/ 50 6 4)
25/12
```

mod

args: number divisor

[Function] returns the root of number modulo divisor. The result will have the same sign as number.

```
> (mod 13 12)
1
```

=

args: number &rest more-numbers
[Function] returns T if all the arguments are numerically equal; otherwise returns NIL.

```
> (= 5 5)
T
> (= 5 5 7)
NIL
```

/=

args: number &rest more-numbers
[Function] returns T if none of the arguments are numerically equal; otherwise returns NIL.

```
> (/= 5 6)
T
```

<

args: number &rest more-numbers
[Function] returns T if each argument is less than the one following it; otherwise returns NIL.

```
> (< 5 9 8)
NIL
> (< 5 5)
NIL
> (< 5 7)
T
> (< 5 6 8)
T
```

>

args: number &rest more-numbers
[Function] returns T if each argument is greater than the one following it; otherwise returns NIL.

```
> (> 5 6 8)
NIL
```

<=

args: number &rest more-numbers

[Function] returns T if each argument is less than or equal to the one following it; otherwise returns NIL.

```
> (<= 5 5)
T
```

>=

args: number &rest more-numbers

[Function] returns T if each argument is greater than or equal to the one following it; otherwise returns NIL.

```
> (>= 5 5 3)
T
```

abs

args: number

[Function] returns the absolute value of number.

```
> (abs -5.5)
5.5
```

truncate

args: number &optional divisor

[Function] returns two values: the integer part of number (i.e. number with the fractional part removed), and the fractional part. When there is a second argument, truncate divides divisor into number first, and then applies truncate to the result.

```
> (truncate 5.8)
5
0.8000002
> (truncate 5.8 2)
1.8000002
```

round

args: number &optional divisor

[Function] returns the integer nearest to number. If number is halfway between two integers (for example 3.5), ROUND converts number to the nearest integer divisible by 2. ROUND returns a second value, which is the remainder of the rounding operation. When there is a second argument, ROUND first divides divisor into number, and then applies ROUND to the result.

```
> (round 14.5)
14
0.5
> (round 15.5)
16
-0.5
> (round 14.5 5)
3
-0.5
```

not

args: object

[Function] returns T if object is NIL; otherwise returns NIL. It inverts its argument considered as a Boolean value.

```
> (not nil)
T
> (not t)
NIL
> (not (equal '(0 1 2 3 4) '(0 1 2 3 4)))
NIL
> (not (equal '(0 1 2 3 4) '(1 2)))
T
> (not (eq 'vilho 'vilho))
NIL
> (not (eq 9 8))
T
```

null

args: thing

[Function] returns T if thing is the empty list (), otherwise returns NIL. This is equivalent to the function NOT, except that NULL is normally used to check for the empty list and NOT to invert. The programmer can express intent by choice of function name.

```
> (null '())
T
```

minusp

args: number

[Function] returns T if number is strictly less than zero; otherwise returns NIL. Number must be a non-complex number.

```
> (minusp -4)
T
```

plusp

args: number

[Function] returns T if number is strictly greater than zero; otherwise returns NIL. Number must be a non-complex number.

```
> (plusp -4)
NIL
```

zerop

args: number

[Function] returns T if number is zero (either the integer zero, a floating-point zero, or a complex zero); otherwise returns NIL. (zerop -0.0) is always true.

```
> (zerop 0)
T
```

numberp

args: object

[Function] returns T if object is a number; otherwise returns NIL. More specific numeric data type tests include **integerp**, **rationalp**, **floatp**, and **complexp**.

```
> (numberp 5)
T
> (numberp 'heikki)
NIL
```

typep

args: thing type

[Function] returns T if thing is of type type; otherwise returns NIL.

```
> (typep 5 'fixnum)
T
> (typep 5.8 'fixnum)
NIL
> (typep 5.8 'float)
T
> (typep 5 'float)
NIL
```

atom

args: data-object

[Function] returns T if object is not a cons; otherwise returns NIL. In general, ATOM is true of anything that is not a list. The one exception is the empty list, which is both a list and an atom.

```
> (atom 4)
T
> (atom 'vilho)
T
> (atom '(0 1 2 3 4 5))
NIL
```

listp

args: object

[Function] returns T if object is a cons or the empty list; otherwise returns NIL. LISTP returns T on the empty list. This is the only difference between LISTP and CONSP. LISTP does not check whether the list is terminated by NIL or is a dotted list.

```
> (listp '(0 1 2 3 4 5))
T
> (listp 4)
NIL
> (listp 'vilho)
NIL
```

eq

args: object1 object2

[Function] returns T if and only if object1 and object2 are the same object. EQ is the fastest and strictest test for equality. (EQ works by testing whether object1 and object2 address the same location in memory.) Things that print the same are not necessarily EQ, numbers with the same value need not be EQ, and two similar lists are usually not EQ.

```
> (eq 'vilho 'vilho)
T
> (eq 9 9)
T
> (eq '(8 9) '(8 9))
NIL
```

equal

args: object1 object2

[Function] returns T when object1 and object2 are structurally similar. A rough rule of thumb is that objects are EQUAL when their printed representation is the same. EQUAL is case sensitive when comparing strings and characters.

```
> (equal '(0 1 2 3 4) '(0 1 2 3 4))
T
> (equal '(0 1 2 3 4) '(1 2))
NIL
```

subsetp

args: list1 list2 &key :test :test-not :key

[Function] returns T if every element of list1 matches some element of list2.

```
> (subsetp '(0 1 2) '(55 3 2 4 0 1 2))
T
> (subsetp '(0 1 6) '(55 2 4 0 1 2))
NIL
```

and

args: {form}*

[Macro] evaluates each form sequentially. If AND reaches a form that returns NIL, it returns NIL without evaluating any more forms. If it reaches the last form, it returns that form's value.

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

or

args: {form}*

[Macro] evaluates each form sequentially. If OR reaches a form that returns non-NIL, it returns the value of that form without evaluating any more forms. If it reaches the last form, it returns that form's value.

```
> (or t t t)
T
> (or () t t)
T
> (or () () ())
NIL
> (not (and t () t))
T
```

set

args: symbol value

[Function] assigns value to the result of evaluating symbol, the name of a dynamic (special) variable. Returns symbol. The function set cannot alter a local (lexically bound) variable.

```
> (set 'foo '(1 2 3))
(1 2 3)
> foo
(1 2 3)
```

setq

args: {variable form}*

[Special Form] sets the value of the current binding of each variable to the result of evaluating the corresponding form. The assignments are performed sequentially. Returns the value of the last variable.

```
> (setq foo '(1 2 3))
(1 2 3)
> foo
(1 2 3)
```

setf

args: {place newvalue}*

[Macro] stores the result of evaluating newvalue into the location that results from examining place. If multiple place-newvalue pairs are specified, they are processed sequentially. SETF returns the last newvalue.

```
> (setf foo '(1 2 3))
(1 2 3)
> foo
(1 2 3)
> (setf (nth 1 foo) 876876)
876876
> foo
(1 876876 3)
```

defvar

args: variable-name &optional initial-value documentation

[Macro] proclaims variable-name to be a special variable, optionally sets it to the value of initial-value, and returns variable-name. If initial-value is given, variable-name is initialized to the result of evaluating it unless variable-name already has a value. If initial-name is not used, it is not evaluated. The macro DEFVAR only has an effect the first time it is called on a symbol. Documentation may be provided as a string. Notice the convention of using stars (*...*) around a global variable name.

```
> (defvar *list* '(555 333 22))
*list*
> *list*
(555 333 22)
```

defparameter

args: variable-name initial-value &optional documentation

[Macro] proclaims variable-name to be a special variable, sets it to the value of evaluating initial-value, a form, and returns variable-name. Documentation may be provided as a string.

```
> (defparameter *list* '(333 22))
*list*
> *list*
(333 22)
```

let

args: ({variable | (variable value) }*) {declaration}* {form}*

[Special Form] creates a binding for each variable (in parallel) and evaluates forms in the resulting environment. Returns the value of the last form.

```
> (let ((val 4)
       (foo)
       (nuotit '(60 55 66 44)))
       (print val)
       (print foo)
       (print nuotit)
       nil)
4
NIL
(60 55 66 44)
NIL
```

car
args: list
[Function] returns the first element of list.

```
> (car '(0 1 2 3 4))
0
```

cdr
args: list
[Function] returns all of list but the first element.

```
> (cdr '(0 1 2 3 4))
(1 2 3 4)
```

caar
args: list
[Function] returns the CAR of the CAR of list.

```
> (caar '((5 6 7) (6 7 8)))
5
```

cddr
args: list
[Function] returns the CDR of the CDR of list.

```
> (cddr '((5 6 7) (6 7 8) (1 2 3)))
((1 2 3))
```

cadar
args: list
[Function] returns the CAR of the CDR of the CAR of list.

```
> (cadar '((5 6 7) (6 7 8) (1 2 3)))
6
```

first

args: list
[Function] returns the CAR of list, using one-based addressing.

```
> (first '(0 1 2 3 4))
0
```

See also :

```
> (second '(0 1 2 3 4))
1
> (third '(0 1 2 3 4))
2
> (fourth '(0 1 2 3 4))
3
> (fifth '(0 1 2 3 4))
4
> (tenth '(0 1 2 3 4))
NIL
```

nth

args: n list"
[Function] returns the NTH element of list (where the CAR of list is the 'zeroth' element).

```
> (nth 2 '(0 1 2 3 4))
2
```

last

args: list &optional count
[Function] returns the last count conses of list.

```
> (last '(0 1 2 3 4))
(4)
> (car (last '(0 1 2 3 4)))
4
> (last '(0 1 2 3 4) 2)
(3 4)
```

subseq

args: sequence start &optional end

[Function] returns a new sequence which contains the elements of the portion of sequence specified by start and end. SUBSEQ may be used with SETF to destructively replace a portion of a sequence.

```
> (subseq '(0 1 2 3 4 5 6 7) 2)
(2 3 4 5 6 7)
> (subseq '(0 1 2 3 4 5 6 7) 0 3)
(0 1 2)
> (subseq '(0 1 2 3 4 5 6 7) 2 3)
(2)
```

butlast

args: list &optional num

[Function] copies all of list except the last num elements, and returns the new list. The num argument defaults to 1. If list is shorter than num, the empty list is returned. The list argument is not modified.

```
> (butlast '(0 1 2 3 4 5 6 7))
(0 1 2 3 4 5 6)
> (butlast '(0 1 2 3 4 5 6 7) 4)
(0 1 2 3)
```

nthcdr

args: n list

[Function] performs the CDR operation n times on list and returns the result.

```
> (nthcdr 3 '(0 1 2 3 4 5 6 7))
(3 4 5 6 7)
```

make-list

args: size &key :initial-element

[Function] returns a list containing size elements, each of which is initialized to :initial-element. size should be a non-negative integer; the default value of :initial-element is NIL.

```
> (make-list 7)
(NIL NIL NIL NIL NIL NIL NIL)
> (make-list 7 :initial-element 66)
(66 66 66 66 66 66 66)
> (make-list 7 :initial-element '(2 3 4))
((2 3 4) (2 3 4) (2 3 4) (2 3 4) (2 3 4) (2 3 4) (2 3 4))
```

append

args: &rest lists

[Function] concatenates the top-level elements of lists, in effect splicing them together. The lists are not modified. Returns the resulting concatenated list.

```
> (append '(0 1 2) '(2 3 4))
(0 1 2 2 3 4)
> (append '(0 1 2) '(2 3 4) '(3 1 22))
(0 1 2 2 3 4 3 1 22)
```

list

args: &rest arguments

[Function] constructs and returns a list containing arguments as its elements.

```
> (list 3 4 5)
(3 4 5)
> (list 3 '(7 8 9) 4 5)
(3 (7 8 9) 4 5)
```

cons

args: x list-or-thing

[Function] allocates a new cons cell whose car is x and whose cdr is list-or-thing.

```
> (cons 3 '(4 5 6))
(3 4 5 6)
```

push

args: item place

[Macro] conses item onto the list contained in place, which can be any generalized variable containing a list and acceptable as a generalized variable to SETF. Stores the resulting list in place and returns the new contents of place.

```
> (setq foo '(1 2 3))
(1 2 3)
> (push 0 foo)
(0 1 2 3)
> foo
(0 1 2 3)
```

pop

args: place

[Macro] returns the CAR of the contents of place, which can be any generalized variable containing a list and acceptable as a generalized variable to SETF. Sets place to point to the CDR of its previous contents.

```
> (pop foo)
0
> foo
(1 2 3)
> (pop foo)
1
> foo
(2 3)
```

`length`

args: sequence

[Function] returns the number of elements in sequence.

```
> (length '(0 1 2 3 4))
5
```

`reverse`

args: sequence

[Function] returns a new sequence with the elements of sequence reversed. The original sequence is not modified.

```
> (reverse '(4 5 6 7 8 9))
(9 8 7 6 5 4)
> (reverse '(4 5 (6 8 9) 7 8 9))
(9 8 7 (6 8 9) 5 4)
```

`member`

args: item list &key :test :test-not :key

[Function] searches list for a top-level element that matches item. If a match is successful, member returns the rest of the list starting with the element that matched item; otherwise returns NIL.

```
> (member 4 '(1 4 5 2))
(4 5 2)
> (member '(4 5 2) '(0 (4 5 2) 2 3 4))
NIL
> (eq '(4 5 2) '(4 5 2))
NIL
> (equal '(4 5 2) '(4 5 2))
T
> (member '(4 5 2) '(0 (4 5 2) 2 3 4) :test #'equal)
((4 5 2) 2 3 4)
> (member '(0 1 2)
          '((56 59 62) (60 61 62) (89 59 62))
          :test #'equal
          :key #'(lambda (l)
                          (mapcar #'(lambda (n)
                                             (mod n 12)) l)))
((60 61 62) (89 59 62))
```

remove

args: item sequence &key :count :start :end :from-end :test :test-not :key

[Function] returns a new sequence equivalent to sequence with occurrences of item removed. The original sequence is not modified. (The destructive counterpart of remove is delete.)

```
> (remove 4 '(3 6 5 4 4 9 4))
(3 6 5 9)
> (remove '(6 7)
        '((5 6 7) (6 7) (5 6 7 5) (6 7))
        :test #'equal)
((5 6 7) (5 6 7 5))
> (remove 4 '(3 6 5 4 4 9 4) :count 1)
(3 6 5 4 9 4)
> (remove 4 '(3 6 5 4 4 9 4) :count 1 :start 5)
(3 6 5 4 4 9)
```

remove-if

args: test sequence &key :from-end :start :end :count :key

[Function] returns a sequence equivalent to sequence except that elements in the given range that pass test are removed. The original sequence is not modified.

```
> (remove-if #'oddp '(0 1 2 3 4 5 6 7 8))
(0 2 4 6 8)
> (remove-if #'evenp '(0 1 2 3 4 5 6 7 8))
(1 3 5 7)
```

subst

args: new old tree &key :test :test-not :key

[Function] creates a new tree based on tree, except that occurrences of old have been replaced with new. The original tree is not modified, but the new tree may share list structure with it.

```
> (subst 'new 'old '(1 2 3 old 4 33 old 4565 old))
(1 2 3 NEW 4 33 NEW 4565 NEW)
```

count

args: item sequence &key :start :end :from-end :key :test :test-not
[Function] returns the number of elements of sequence that match item using the given test function; returns nil if no element matches.

```
> (count 6 '(0 1 2 6 2 3 6 6))
3
> (count 6 '(48 67 66 54 81 90))
0
> (count 6 '(48 67 66 54 81 90) :key #'(lambda (n) (mod n 12)))
3
```

count-if

args: test sequence &key :from-end :start :end :key
[Function] returns the number of elements in the given range of sequence that satisfy test.

```
> (count-if #'oddp '(0 1 2 3 4 5 6 7 8))
4
> (count-if #'evenp '(0 1 2 3 4 5 6 7 8))
5
```

dotimes

args: (var countform [resultform]) {declaration}* {tag | statement}*

[Macro] executes forms countform times. On successive executions, var is bound to the integers between zero and countform. Upon completion, resultform is evaluated, and the value is returned. If resultform is omitted, the result is NIL.

```
> (dotimes (luku 3) (print luku))
0
1
2
NIL
```

do

args: ({var | (var [init [step]])}*) (end-test {result}*) {declaration}* {tag | statement}*

[Macro] at the beginning of each iteration, evaluates all the init forms (before any var is bound), then binds each var to the value of its init . Then evaluates end-test; if the result is NIL, execution proceeds with the body of the form. If the result is non-NIL, the result forms are evaluated as an implicit PROGN and the value of the last result form is returned. At the beginning of the second and subsequent iterations, all step forms are evaluated, then all variables are updated.

```
> (do ((i 0 (+ i 1)))
      ((= i 10))
      (print i))
0
1
2
3
4
5
6
7
8
9
nil

> (do ((i 0 (+ i 1))(j 0 (+ j 0.5)))
      ((or (> j 2.24) (= i 10)))
      (print (list i j)))
(0 0)
(1 0.5)
(2 1.0)
(3 1.5)
(4 2.0)
```

dolist

args: (var listform [resultform]) {declaration}* {tag | statement}*

[Macro] evaluates listform, which produces a list, and executes the body once for every element in the list. On each iteration, var is bound to successive elements of the list. Upon completion, resultform is evaluated, and the value is returned. If resultform is omitted, the result is NIL.

```
> (let (res (lista '(0 1 2 3 4 3 2 1)))
    (dolist (item lista (nreverse res))
            (push (1+ item) res)))
(1 2 3 4 5 4 3 2)
```

cond

args: {(test {form}*)}*

[Macro] consists of a series of clauses which are tested sequentially. If a test is T, COND evaluates the corresponding forms and returns the last form's value. If the test returns NIL, cond proceeds to the next test / form clause. If all tests fail, cond returns nil.

```
> (defun test-nums (num1 num2)
      (cond ((= num1 6) 'first-test)
            ((= num2 7) 'second-test)
            ((= num1 num2) 'third-test)
            (t 'last-test)))
test-nums
> (test-nums 6 7)
first-test
> (test-nums 87 87)
third-test
```

if

args: testform thenform [elseform]

[Special Form] evaluates testform. If the result is T, evaluates thenform and returns the result; if the result is NIL, evaluates elseform and returns the result.

```
> (defun test-nums2 (num1 num2)
      (if (= num1 num2) 'same 'different))
test-nums2
> (test-nums2 6 7)
different
> (test-nums2 6 6)
same
```

when

args: testform {thenform}*

[Macro] evaluates testform. If the result is NIL, returns NIL without evaluating any thenform. If the result is T, evaluates thenforms as an implicit PROGN, sequentially from left to right, and returns the value of the last thenform.

```
> (when (= 5 5)
      (print 'foo)
      (+ 5 6))
FOO
11
```

unless

args: testform {thenform}*

[Macro] evaluates testform. If the result is non-NIL, then no thenforms are evaluated and UNLESS returns NIL. If the result is NIL, evaluates thenforms as an implicit PROGN, sequentially from left to right, and returns the value of the last thenform.

```
> (unless (= 5 5)
      (print 'foo)
      (+ 5 6))
NIL
> (unless (= 5 99)
      (print 'foo)
      (+ 5 6))
FOO
11
```

case

args: keyform {({({key }*) | key } {form}*) }*

[Macro] evaluates keyform, then evaluates as an implicit PROGN the forms whose keys match the value of keyform. Returns the last form evaluated. keyform is evaluated, but the keys are not. CASE permits a final case, otherwise or T, that handles all keys not otherwise covered.

```
> (case 5
      (3 (+ 3 4))
      (5 (print 'second) (+ 55 4))
      (11 (print 'third) (- 3 4))
      (23 (+ 3 9))
      (t (+ 3 4)))
second
59
```

eval

args: form

[Function] evaluates form and returns the value returned by form. The evaluation is performed in the current dynamic environment and a null lexical environment.

```
> (eval '(+ 5 6))
11
> '(+ 5 6)
(+ 5 6)
```

apply

args: function first-arg &rest more-args

[Function] invokes function, giving it first-arg and more-args as arguments. The value returned by function is returned. The last argument to APPLY should be a list; the elements of this list are passed as individual arguments to function. The type of function can be only symbol or function.

```
> (apply 'list '(5 6 7 g b))
(5 6 7 G B)
> (apply '+ '(5 6 7 5.4 6.7))
30.099999999999998
> (setq plussa '+)
+
> (apply plussa '(5 6 7 5.4 6.7))
30.099999999999998
```

funcall

args: function &rest arguments

[Function] invokes function, passing it arguments as arguments. Because FUNCALL is a function, function is evaluated. The type of function can be only symbol or function. The value returned by the function call is returned.

```
> (funcall '+ 4 5)
9
> (funcall plussa 4 5)
9
> (funcall #'(lambda (a b) (+ a b)) 5 6)
11
```

mapcar

args: function list &rest more-lists

[Function] applies function to the CAR of list and more-lists, then to the CADR, and so on. The results are collected into a list, which is returned. If the lists are not all the same length, the iteration terminates when the shortest list runs out. function can be only of type symbol or function.

```
> (mapcar #'abs '(3 4 -3 2 -22 5))
(3 4 3 2 22 5)
> (mapcar #'+ '(3 4 -3 2 -22 5) '(3 4 -3 2 -22 5))
(6 8 -6 4 -44 10)
> (mapcar #'list '(3 4 -3 2 -22 5))
((3) (4) (-3) (2) (-22) (5))
> (mapcar #'+
          '(3 4 -3 2 -22 5) '(3 4 -3 2 -22 5) '(3 4 -3 2 -22 5))
(9 12 -9 6 -66 15)
> (mapcar #'+ '(3 4 -3 2 -22 5) '(1 1 1))
(4 5 -2)
> (mapcar #'(lambda (a) (+ a 5)) '(3 4 -3 2 -22 5))
(8 9 2 7 -17 10)
```

quote

args: object

[Special Form] returns object , which may be any object, without evaluating it.

```
> (quote uolevi)
uolevi
```

sort

args: sequence predicate &key :key

[Function] destructively sorts the elements of sequence into an order determined by predicate.

```
> (sort '(77 5 6 33 7 8) '<)
(5 6 7 8 33 77)
> (sort '(77 5 6 33 7 8) '>)
(77 33 8 7 6 5)
> (sort '(vilho paavo aatami jussi) 'string-lessp)
(AATAMI JUSSI PAAVO VILHO)
> (defun sort-chords (chords)
      (sort chords '<
            :key #'(lambda (chord)
                         (apply #'+
                               (mapcar #'- (cdr chord) chord)))))
> (sort-chords '((0 3 7) (5 6 7) (0 6 11)))
((5 6 7) (0 3 7) (0 6 11))
```

adjoin

args: item list &key :test :test-not :key

[Function] adds item to list if it is not already a member of list, and returns the resulting list. list is not destructively modified.

```
> (adjoin 99 '(0 1 2 3 4 5 6 7 8))
(99 0 1 2 3 4 5 6 7 8)
> (adjoin 6 '(0 1 2 3 4 5 6 7 8))
(0 1 2 3 4 5 6 7 8)
```

union

args: list1 list2 &key :test :test-not :key

[Function] returns a list containing the union of the elements list1 and list2. Any element that is contained in list1 or list2 will be contained in the result list. If there are duplicate entries, only one will appear in the result. list1 and list2 are not modified.

```
> (union '(4 3 2 5) '(7 1 2 5 4 3))
(7 1 2 5 4 3)
```

set-difference

args: list1 list2 &key :test :test-not :key

[Function] returns the list of elements of list1 that are not elements of list2. list1 and list2 are not modified.

```
> (set-difference '(0 1 2 3 4) '(2 3 7 1 2 3))
(4 0)
```

intersection

args: list1 list2 &key :test :test-not :key

[Function] returns the intersection of list1 and list2, that is, a list of those elements which are in both list1 and list2. If either list has duplicate entries, the redundant entries may or may not appear in the result. list1 and list2 are not modified.

```
> (intersection '(0 1 2 3 4) '(3 4 5 6 7))
(4 3)
```