OpenLisp v11.6.0

Reference Manual

By C. Jullien - Eligis
4, villa des REINETTES
95390 Saint-Prix
France

Last modification 2022/09/06

CONTENTS

1 Introduction

OpenLisp is a KISS (Keep It Small and Simple) full conforming implementation of *ISO/IEC 13816:2007 ISLISP Language*, the International Standard version of Lisp. Entirely written in C, **OpenLisp** is essentially a very fast interpreter that competes in speed with some CLtL compilers. It also has an incremental compiler that generates portable LAP code across supported platforms. It is written in ISO C for the kernel and using POSIX like interface for the operating System when available. In the usual case, a new UNIX port is as simple as "make POSIX".

The goal of **OpenLisp** is to provide an efficient, modern and complete Lisp System for those whom want embedded Lisp processing in more conventional applications written in C, C++ or even Visual Basic. Even if **OpenLisp** can be used with a toplevel loop and with all goodies that an old-timer lisp user enjoy, it is more tailored to be transparently integrated in a native C or C++ applications. For this purpose, **OpenLisp** is distributed mainly as a Lisp library (or DLL on Microsoft world) that you can integrate into your main application. The memory footprint is very small with less than 200 Kbytes for the complete kernel and less than 400 Kbytes for a usable Lisp System data. There is no limit on the maximum memory that the System can use. With little efforts, you can exchange data between C and Lisp. **OpenLisp** extends the ISLISP standard to ease port from other Lisp Language, mainly Common Lisp. The kernel can be compiled to support the IS0/IEC 10646-1 (16 bits) character sets instead of the IS0 8859-1 (8 bits) character sets. It also provides a consistent interface to communicate using Lisp streams with BSD, POSIX sockets and/or WinSocks sockets or distributed architectures using DCOM or CORBA.

2 Presentation

LISP is one of the oldest programming languages, invented in 1960 by John McCarthy in order to write programs for Artificial Intelligence easily. **OpenLisp** is the first LISP specially designed to fully comform the international standard *ISO/IEC 13816:2007(E) : Programming language ISLISP*. This standard is the result of cooperative efforts by the design committee.

The following factors influenced the establishment of design goals for ISLISP:

A desire of the international LISP community to standardize on those features of LISP upon which there is widespread agreement.
The existence of the incompatible dialects COMMON-LISP, EULISP, LE-LISP, and SCHEME (mentioned in alphabetical order).
A desire to affirm LISP as an industrial language.
This led to the following design goals for ISLISP:

ISLISP shall be compatible with existing LISP dialects where feasible.
ISLISP shall have as a primary goal to provide basic functionality.
ISLISP shall be object-oriented.
ISLISP shall be designed with extensibility in mind.
ISLISP shall give priority to industrial needs over academic needs.
ISLISP shall promote efficient implementations and applications.

OpenLisp meets all the above goals. Since **OpenLisp** is not a package added to an existing Lisp, the first source for its documentation should be *ISO/IEC 13816:2007(E): Programming Language ISLISP* or the equivalent Public Domain definition that comes with this documentation. To extend the goal of compatibility with other Lisp implementations, **OpenLisp** has extended the ISLISP standard to provide more functions that will help portability across different Lisp System. As much as possible, those functions should be avoided if you want the maximal portability. This document describes only added functions to ISLISP Standard.

3 Portability.

OpenLisp kernel is entirely written in C using, when available, only *ISO/IEC 9899:1990 C Programming Language* features and functions. Using proper directives in header files, it can also be transparently integrated with C++. It supports also older compilers (read K&R). Input/output routines rely on C ISO (*C Standard Language-Dependent System Support*) and for advanced features on UNIX, ISO/IEC 9945-1:1990 (*Strictly Conforming POSIX.1 Application*). The system depend features are grouped in file **`physio.c`** (PHYSical I/O) it can be optimized on a specific system. For example, NT port uses the Virtual Memory allocator. The Garbage Collector, a classic mark & sweep, is also written in C and tests, at runtime (or sometimes at compile time) the features of the underline hardware (stack, alignments, virtual memory, segmented memory, threads…).

OpenLisp has been ported on almost all modern processors from 16 to 64 bits architecture (Intel 16/32 bits, Intel Itanium, Sparc 32 bits et UltraSparc 64 bits, Motorola 68k and 88k, RS6000, PowerPC, MIPS R3x00, R4x00, R10000 32/64 bits, HP PA, Alpha mode 32/64 bits, ARM, ARM64,Vax, IBM MVS).

On all those systems, the compiler has been configured to produce the maximum level of warnings and **OpenLisp** is still warning free!! That way, a new port of **OpenLisp** is very easy. Typically, on new Unix system, it as easy as just:

```
./configure; make
```

All the kernel functions are written in C, only environment functions like pretty, sort, and useful macros are written in Lisp. **OpenLisp** load a file named **`startup.lsp`** that, in turns, defines most of other packages as autoload features.

4 Credits.

OpenLisp is an original development that uses two optional libraries:

- BigNum package developed jointly by INRIA and Digital PRL.
- Regular expressions package based on Henry Spencer regexp source code.

5 Getting started

Internally, **OpenLisp** has 6 different zones (cons, symbol, string, vector, float, heap) to store its objects. Each zone is configurable by 4 Kb page chunk. When you launch **OpenLisp**, it uses some « standard » size which may change on System and/or processor type. On plain old MS-DOS it will not be the same as OSF1 using an Alpha 64 bits processor. The **`room`** Lisp function may be used to fix the total amount of memory you need for each zone. Generally, float and integer numbers are always coded in the address of the object, so you can leave the zone empty (*i.e.* 0).

On most modern systems (Windows and above, nearly all unix implementations), **OpenLisp** can allocate memory using virtual memory routines. It means that you generally don't care to specify how much cons, symbol, string, vector, float or heap you need. The memory zones can grow automatically. For systems with virtual memory, you can reserve a minimal number of mega-bytes for page objects and heap. Note that *reserve* does not mean *allocate*.

To change the default startup size for each zone, you can use the following options (remember that the size you give allocate a 4 Kb page for the object). Note also that, since a symbol has also a print name, the string zone must be greater than the symbol zone.

Set startup zones:

`--cons`	number of 4 Kb pages for **`<cons>`** zone	(2 pointers)
`--symbol`	number of 4 Kb pages for **`<symbol>`** zone	(8 pointers)
`--string`	number of 4 Kb pages for **`<string>`** zone	(2 pointers)
`--vector`	number of 4 Kb pages for vectors, objects, arrays zones	(2 pointers)
`--float`	number of 4 Kb pages for **`<float>`** zone	(0/2 pointers)
`--heap`	number of Kb to store internal representation of objects.	

All options:

--eval/-e *expr*	run *expr* and exit.
--rational	force use of rational numbers.
--norational	don't use rational numbers.
--bf	floats are boxed and internal representation is a **double**.
--uf	floats are unboxed and internal representation is either 31 or 63bits.
--emacs	allows **OpenLisp** to run as GNU Emacs **lisp-inferior-mode.**
--keep	keep **OpenLisp** running when a file is given as an argument.
--last	print the last result when a file is given as an argument.
--islisp	enforce ISLISP behavior.
--noinit	don't load startup files.
--novm	don't use virtual allocation routine, force standard malloc.
--odsp	OpenLisp Dynamic Server Page mode loading.
--quiet	quiet mode loading.
--shell	shell mode loading.
--quit	exit with code 0 after option processing.
--disable-debugger	disable debugger and exit on error.
--utf8	utf8 terminal input/output.
--version	displays **OpenLisp** version and exit.
--vheap	minimal number of Mb that VM reserves for heap.
--vpage	minimal number of Mb that VM reserves for objects pages.
--vhratio	heap ratio from VM page size. Ex: 40 means 40% of VM page size.
-D	run as daemon (see **run-as-daemon** function).
--	next arguments are passed to user's program.

```
% openlisp -cons 100 -symbol 32 -string 48 -vector 32 -heap 512
```

On Windows and most unix systems, the **heap** grows dynamically as needed. This way, you can start with a minimal zone (*i.e.*: 30 Kb) that will expand during execution.

On nearly all systems, **OpenLisp** can save memory images that can restored at any time (see the definition of **save-core** and **restore-core**). When a core image has been saved, you can restore it directly from command line with the **-r** option.

-r	restore a core image.

```
% openlisp -r image.cor
```

You can also load and/or execute a lisp file at startup by adding the file as command line argument.

```
% openlisp demo/myfile.lsp
```

You can install **OpenLisp** anywhere on the System. By default, **OpenLisp** will search its files in the following directories in order: **lib**, **fsl**, **bench** and **contrib** are sub-directories of the current directory. If **OpenLisp** has been installed in **/lisp** directory you must have the following tree:

```
/lisp/bench
/lisp/contrib
/lisp/lib
/lisp/net
```

If you want to launch **OpenLisp** anywhere you must add **OpenLisp** directory to your PATH and set the environment variable named **OPENLISP** install directory.

```
% set OPENLISP=/lisp
```

6 Language Extensions to ISLISP Standard

7 ISLISP Compatibility.

OpenLisp can issue warnings when ISLISP semantic extensions are used (function redefinition, setq at toplevel…). By default, (warning level 0) no warnings are displayed.

`*warning-level*`	**`dynamic-variable`**

`*warning-level*` dynamic variable (default value 0) set the level of warning raised by the system. Current values are 0 for no warnings and 1 for ISLISP extensions used.

8 Controlling the reader and the printer.

OpenLisp is case insensitive by default but we can change this behavior using the **`preserve-read-case`** function.

`(preserve-case-flag` *flag* **`)`** -> <*boolean*>	**`macro`**

`(preserve-case-flag t)` don't change the character case to a neutral character representation. With this mode, **`Foo`** and **`FOO`** are two different symbols. Called with **`nil`**, this function restores the standard neutral convention. With no argument at all, it returns the current value in use.

`*prompt*`	**`dynamic variable`**

`*prompt*` dynamic variable can be used to control the prompt reader.

Example:

```
? (dynamic-let ((*prompt* "Your guess> "))
          (read))
Your guess> 10
= 10
?
```

system-path -> *<list>* **dynamic variable**

This variable is set to the directories list used by **OpenLisp** when trying to load a library file.

load-verbose -> *<boolean>* **dynamic variable**

When **t**, each loaded files are printed on to console.

read-base **dynamic variable**

The dynamic value of ***read-base*** controls the interpretation of tokens by read as being integers. Its value is the radix in which integers are to be read; the value may be any integer from `2` to `36` (inclusive) and is normally `10` (decimal radix). Its value affects only the reading of integers.

read-suppress **dynamic variable**

This dynamic variable is intended primarily to support the operation of the read-time conditional notations **#+** and **#-**. If it is **nil**, the Lisp reader operates normally. If the value of ***read-suppress*** is **t**, **read**, **read-delimited-list**, and **read-from-string** all return **nil** when they complete successfully; however, they continue to parse the representation of an object in the normal way, in order to skip over the object, and continue to indicate end of file in the normal way. Except as noted below, any standardized reader macro that is defined to read a following object or token will do so, but not signal an error if the object read is not of an appropriate type or syntax.

Example:

```
? (dynamic-let ((*read-suppress* t))
          (read-from-string "(1 . 2 . 3)") ;; read without error
= nil
?
```

print-base **dynamic variable**

The dynamic value of ***print-base*** determines in what radix the printer will print rationals. This may be any integer from `2` to `36`, inclusive; the default value is `10` (decimal radix). For radices above `10`, letters of the alphabet are used to represent digits above `9`.

print-radix **dynamic variable**

The dynamic value of ***print-radix*** (default **false**) controls the printing of rationals. If the value of ***print-radix*** is **true**, the printer will print a radix specifier to indicate the radix in which it is printing a rational number. The radix specifier is always printed using lowercase letters. If ***print-base*** is 2, 8, or 16, then the radix specifier used is **#b**, **#o**, or **#x**, respectively.

Example:

```
? (for ((i 2 (1+ i)))
       ((> i 36))
       (dynamic-let ((*print-base* i)
                     (*print-radix* t))
```

```
                (format (standard-output) "~a~%" 76876786/27865)))
#b100100101010000101111110010/110110011011001
#3r12100122202010101/1102020001
#4r10211100233302/12303121
#5r124140024121/1342430
#6r11343423014/333001
#7r1622304400/144145
#o445205762/66331
#9r170582111/42201
#10r76876786/27865
#11r3a438738/19a32
#12r218b4a6a/14161
#13r12c0892c/c8b6
#14ra2d2470/a225
#15r6b38491/83ca
#x4950bf2/6cd9
#17r3327aaf/5b72
#18r24c5g0a/4e01
#19r1c0h31c/413b
#20r1409bj6/39d5
#21rih62g7/303j
#22rek3i78/2dcd
#23rblgagm/26fc
#24r9fh2fa/2091
#25r71k21b/1jef
#26r6c5p1c/1f5j
#27r59hk3a/1b61
#28r4d213e/17f5
#29r31k34j/143p
#30r34r8jg/10sp
#31r217gka/sur
#32r29a2vi/r6p
#33r1vr6uj/pjd
#34r1nhwdw/o3j
#35r1g81jl/mq5
#36r19rqia/li1
```

read-level **dynamic variable**

When ***read-level*** dynamic variable is not **nil**, (the default) the reader prints a string showing the current read level.

Example:

```
? (defun foo ()
1>   (progn
2>      (print 'foo)
2>      t)))
= foo
?
```

print-escape **dynamic variable**

When this dynamic variable is **nil**, then escape characters are not output when an expression is printed.

print-nil-as-list **dynamic variable**

print-nil-as-list dynamic variable can be used to control how **nil** is printed. By default, **nil** is printed as a symbol *(i.e. using 3 letters* **'n' 'i' 'l'***)*. If the value of this dynamic variables is **t**, **nil** is printed as the empty list *(i.e.* **()***)*.

Example:

```
? nil
= nil
? (setf (dynamic *print-nil-as-list*) t)
= t
? nil
= ()
? (setf (dynamic *print-nil-as-list*) nil)
= nil
? nil
= nil
```

`*print-level*` **dynamic variable**

`print-level` dynamic variable can be used to limit the depth for which an expression is printed.

Example:

```
? (dynamic-let ((*print-level* 3))
     (print '(1 (2 (3 (4 (5))))))
     t)
(1 (2 (... ... )))
= t
```

`*print-length*` **dynamic variable**

`print-length` dynamic variable can be used to limit the length for which an expression is printed.

Example:

```
? (dynamic-let ((*print-length* 3))
     (print '(1 2 3 4 5))
     t)
(1 2 3 ... )
= t
```

`*print-package*` **dynamic variable**

`print-package` dynamic variable, when non-**`nil`**, directs the Lisp printer to show the package of each printed symbols.

Example:

```
? (dynamic-let ((*print-package* t))
     (print '(car system :test foo))
     t)
(islisp:car openlisp:system keyword:test user:foo)
= t
```

`*last-error*` -> *<condition>* **dynamic variable**

`*last-error*` dynamic variable is the last condition (if any) raised by the system.

`*default-encoding*` -> *<symbol>* **dynamic variable**

Define default file encoding. It currentlty supports **<character>**, **<wide-character>** and **<utf8-character>**.

9 Extended Dispatching Macro Character Syntax.

OpenLisp extends standard syntax introduced by the **#** character. These take the general form of a **#**, a second character that identifies the syntax, and following arguments in some form. If the second character is a letter, then case is not important; **#A** and **#a** are considered to be equivalent.

`#s( .. )`	defines a structure (see **defstruct**)
`#+, #-`	read-time conditional
`#.`	read-time evaluation
`#!`	line comment, useful as shell extension
`#&`	FASL reference (see FASL)

#s The syntax **#s(***name slot1 value1 slot2 value2* **...)** denotes a structure. This is legal only if *name* is the name of a structure already defined by **defstruct** and if the structure has a standard constructor macro, which it normally will.

#+ The **#+** syntax provides a read-time conditionalization facility; the syntax is

#+*feature form*

If *feature* is "true", then this syntax represents a Lisp object whose printed representation is *form*. If *feature* is "false", then this syntax is effectively whitespace; it is as if it did not appear.

The rules for interpreting a feature expression are as follows:

feature

If a symbol naming a feature is used as a feature expression, the feature expression succeeds if that feature is present; otherwise it fails.

(not *feature-conditional***)**

A not feature expression succeeds if its argument feature-conditional fails; otherwise, it succeeds.

(and *feature-conditional****)**

An **and** feature expression succeeds if all of its argument feature-conditionals succeed; otherwise, it fails.

(or *feature-conditional****)**

An **or** feature expression succeeds if any of its argument feature-conditionals succeed; otherwise, it fails.

#- **#-***feature form* is equivalent to **#+(not** *feature***)** form.

#! `#! simply ignore` the rest of the line. This syntax is useful when you want to create unix shell in Lisp. Assuming that **openlisp** is in your path, you can execute a file:

```
#!/usr/bin/env openlisp -shell
(format (standard-output) "(fib 20) = ~s~%" (fib 20))
```

that computes some value in Lisp.

#. `#.`*foo* is read as the object resulting from the evaluation of the Lisp object represented by *foo*, which may be the printed representation of any Lisp object. The evaluation is done during the **read** process, when the **#.** construct is encountered.

#n=*obj* The syntax **#*n*=** *obj* reads as whatever Lisp object has *obj* as its printed representation. However, that object is labelled by *n*, a required unsigned decimal integer, for possible reference by the syntax **#*n*#** (below). The scope of the label is the expression being read by the outermost call to **read**. Within this expression the same label may not appear twice.

#n# The syntax **#*n*#**, where *n* is a required unsigned decimal integer, serves as a reference to some object labelled by **#*n*=**; that is, **#*n*#** represents a pointer to the same identical (**eq**) object labelled by **#*n*=**. This permits notation of structures with shared or circular substructure.

#&*n* **#&***n* returns the n^{th} value of a predefined internal vector. This syntax, related to LAP format, should not be used for another purpose. **Now obsolete and may be removed in a future version**.

10 Control of Time of Evaluation.

The `eval-when` special form allows pieces of code to be executed only at compile time, only at load time, or when interpreted but not compiled.

(eval-when (*situation****)** *form****)** -> <*object*> **special form**

The body of an **eval-when** form is processed as an implicit **progn**, but only in the situations listed. Each *situation* must be a symbol, either **compile**, **load**, or **eval**.

eval specifies that the interpreter should process the body. **compile** specifies that the compiler should evaluate the body at compile time in the compilation context. **load** specifies that the compiler should arrange to evaluate the forms in the body when the compiled file containing the **eval-when** form is loaded.

Example:

```
(eval-when (eval compile)
   ;; the following macros are not used once compiled
   (defmacro …)
)
```

11 Control Structures.

(when *test form****)** -> <*object*> **macro**

(when *test* $form_1$ $form_2$ **...)** first evaluates *test*. If the result is **nil**, then no *form* is evaluated, and **nil** is returned. Otherwise the *form*s constitute an implicit **progn** and are evaluated sequentially from left to right, and the value of the last one is returned.

```
(when p a b c) == (and p (progn a b c))
(when p a b c) == (cond (p a b c))
(when p a b c) == (if p (progn a b c) nil)
(when p a b c) == (unless (not p) a b c)
```

As a matter of style, **when** is normally used to conditionally produce some side effects, and the value of the **when** form is normally not used. If the value is relevant, then it may be stylistically more appropriate to use **and** or **if**.

(unless *test form**) -> <object> **macro**

(unless *test form$_1$ form$_2$* **...)** first evaluates *test*. If the result is *not* **nil**, then the *forms* are not evaluated, and **nil** is returned. Otherwise the *forms* constitute an implicit **progn** and are evaluated sequentially from left to right, and the value of the last one is returned.
(unless p a b c) == (cond ((not p) a b c))
(unless p a b c) == (if p nil (progn a b c))
(unless p a b c) == (when (not p) a b c)
As a matter of style, **unless** is normally used to conditionally produce some side effects, and the value of the **unless** form is normally not used. If the value is relevant, then it may be stylistically more appropriate to use **if**.

(do ((*var init* **[***step***])*) (***end-test result**) *body***)** -> <object> **special form**

The **do** special form provides a generalized iteration facility, with an arbitrary number of ``index variables." These variables are bound within the iteration and stepped in parallel in specified ways. They may be used both to generate successive values of interest (such as successive integers) or to accumulate results. When an end condition is met, the iteration terminates with a specified value. In **OpenLisp**, this special form is an alias to the ISLISP **for** special form.

(do* ((*var init* **[***step***])*) (***end-test result**) *body***)** -> <object> **special form**

do* is exactly like **do** except that the bindings and steppings of the variables are performed sequentially rather than in parallel. In **OpenLisp**, this special form is an alias to the ISLISP **for*** special form.

(dolist ((*var listform* **[***resultform***])*)** *body***)** -> <object> **macro**

dolist provides straightforward iteration over the elements of a list. First **dolist** evaluates the form *listform*, which should produce a list. It then executes the body once for each element in the list, in order, with the variable *var* bound to the element. Then *resultform* (a single form, *not* an implicit **progn**) is evaluated, and the result is the value of the **dolist** form. (When the *resultform* is evaluated, the control variable *var* is still bound and has the value **nil**.) If *resultform* is omitted, the result is **nil**.

Example:

```
(dolist (x '(a b c d)) (format t "~s " x)) => nil
   after printing ``a b c d '' (note the trailing space)
```

(dotimes ((*var countform* **[***resultform***])*)** *body***)** -> <object> **macro**

dotimes provides straightforward iteration over a sequence of integers. It evaluates the form *countform*, which should produce an integer. It then performs *progbody* once for each integer from zero (inclusive) to *count* (exclusive), in order, with the variable *var* bound to the integer; if the value of *countform* is zero or negative, then the *body* is performed zero times. Finally, *resultform* (a single form, *not* an implicit **progn**) is evaluated, and the result is the value of the **dotimes** form. (When the *resultform* is evaluated, the control variable *var* is still bound and has as its value the number of times the body was executed.) If *resultform* is omitted, the result is **nil**.

(typecase ((*class-name****)** *form****)* [(t** *form****)])** -> <object> **macro**

typecase is a conditional that chooses one of its clauses by examining the ***class-name*** of an object. Its form is as follows:

```
(typecase keyform
  ((class-name-1) form-1-1 form -1-2 ...)
  ((class-name -2) form -2-1 ...)
  ((class-name -3) form -3-1 ...)
```

```
  ...
  (t consequent-t-1 ...))
```

Structurally **typecase** is much like case, selecting one clause and then executing all consequents of that clause.

(ecase ((<u>*clause*</u>*****)** *form****)*******)** -> *<object>* **macro**

ecase is similar to **case**, but no explicit default clause is permitted. If no clause is satisfied, **ecase** signals an error with a message constructed from the clauses.

(dynamic-let* ((<u>*var*</u> *form***)*)** *body****) ->** *<object>* **special form**

This function works much as **let***, but for dynamic variables.

(prog1 *first rest***)** -> *<object>* **special form**

prog1 is similar to **progn**, but it returns the value of its *first* form. All the argument forms are executed sequentially; the value of the first form is saved while all the others are executed and is then returned.
prog1 is most commonly used to evaluate an expression with side effects and to return a value that must be computed *before* the side effects happen.

<u>Example:</u>

```
(prog1 (car x) (setf (car x) 'foo))
```

alters the *car* of **x** to be **foo** and returns the old *car* of **x**.

(psetq <u>var_1</u> val_1 ... <u>var_n</u> val_n**)** -> *<object>* **special form**

A **psetq** form is just like a **setq** form, except that the assignments happen in parallel. First all of the forms are evaluated, and then the variables are set to the resulting values. The value of the **psetq** form is **nil**. For example:

<u>Example:</u>

```
(setq a 1)
(setq b 2)
(psetq a b  b a)
a => 2
b => 1
```

In this example, the values of **a** and **b** are exchanged by using parallel assignment.

(defsetf <u>*place*</u> *form****)** -> *<object>* **function**

Defines a new place that can be used by **setf** special form.

(incf <u>*place*</u>**)** -> *<integer>* **macro**
(decf <u>*place*</u>**)** -> *<integer>* **macro**

The number produced by the form *delta* is added to (**incf**) or subtracted from (**decf**) the number in the generalized variable named by *place*, and the sum is stored back into *place* and returned. The form *place* may be any form acceptable as a generalized variable to **setf**. If *delta* is not supplied, then the number in *place* is changed by `1`.

<u>Example:</u>

```
(setq n 0)  => 0
(incf n)    => 1  and now n => 1
(decf n 3)  => -2 and now n => -2
(decf n -5) => 3  and now n => 3
(decf n)    => 2  and now n => 2
```

The effect of **(incf place delta)** is roughly equivalent to
(setf place (+ place delta))
except that the latter would evaluate any subforms of *place* twice, whereas **incf** takes care to evaluate them only once. Moreover, for certain *place* forms **incf** may be significantly more efficient than the **setf** version.

(push *item place***)** -> <*object*> **macro**

The form *place* should be the name of a generalized variable containing a list; *item* may refer to any Lisp object. The *item* is consed onto the front of the list, and the augmented list is stored back into *place* and returned. The form *place* may be any form acceptable as a generalized variable to **setf**. If the list held in *place* is viewed as a push-down stack, then **push** pushes an element onto the top of the stack.

Example:

```
(setq x '(a (b c) d))                  => (a (b c) d)
(push 5 (cadr x))                      => (5 b c)
x                                      => (a (5 b c) d)
```

The effect of **(push** *item place***)** is roughly equivalent to **(setf** *place* **(cons** *item place***))** except that the latter would evaluate any subforms of *place* twice, while **push** takes care to evaluate them only once. Moreover, for certain *place* forms **push** may be significantly more efficient than the **setf** version.

(pushnew *item place* [**:test** *test-function*]**))** -> <*object*> **macro**

The form *place* should be the name of a generalized variable containing a list; *item* may refer to any Lisp object. If the *item* is not already a member of the list (as determined by comparisons using the **:test** predicate, which defaults to **eql**), then the *item* is consed onto the front of the list, and the augmented list is stored back into *place* and returned; otherwise the unaugmented list is returned. The form *place* may be any form acceptable as a generalized variable to **setf**. If the list held in *place* is viewed as a set, then **pushnew** adjoins an element to the set; see **adjoin**.

Example:

```
(setq x '(a (b c) d))   => (a (b c) d)
(pushnew 5 (cadr x))    => (5 b c)
x     => (a (5 b c) d)
(pushnew 'b (cadr x))   => (5 b c)
x     => (a (5 b c) d)
```

(pop *place***)** -> <*object*> **macro**

The form *place* should be the name of a generalized variable containing a list. The result of **pop** is the **car** of the contents of *place*, and as a side effect the **cdr** of the contents is stored back into *place*. The form *place* may be any form acceptable as a generalized variable to **setf**. If the list held in *place* is viewed as a push-down stack, then **pop** pops an element from the top of the stack and returns it.

Example:

```
(setq stack '(a b c))   => (a b c)
(pop stack) => a
```

stack => (b c)

12 Evaluation Functions

(eval *form* **[***environment***])** -> <*object*> **function**

The *form* is evaluated in the current dynamic environment and a null lexical environment. Whatever results from the evaluation is returned from the call to **eval**. If an optional 2nd argument is given, it must be an environmment objet as created by **the-environnment** function. In that case, the expression form is evaluated in this lexical environment instead of null lexical environment (this feature is only supported by **OpenLisp**).
Note that when you write a call to **eval** *two* levels of evaluation occur on the argument form you write. First the argument form is evaluated, as for arguments to any function, by the usual argument evaluation mechanism (which involves an implicit use of **eval**). Then the argument is passed to the **eval** function, where another evaluation occurs.

Example:

```
(eval (list 'cdr (car '((quote (a . b)) c)))) => b
```

The argument form **(list 'cdr (car '((quote (***a* **.** *b***))** *c***)))** is evaluated in the usual way to produce the argument **(cdr (quote (***a* **.** *b***)));** this is then given to **eval** because **eval** is being called explicitly, and **eval** evaluates its argument **(cdr (quote (***a* **.** *b***)))** to produce *b*.
If all that is required for some application is to obtain the current dynamic value of a given symbol, the function **symbol-value** may be more efficient than **eval**.

(the-environment) -> <*environment*> **function**

Returns an object of type **<environment>** that contains the current lexical closure. This object can be passed as the optional second argument of **eval**.

(constantp *object***)** -> <*boolean*> **function**

If the predicate **constantp** is true of an object, then that object, when considered as a form to be evaluated, always evaluates to the same thing; it is a constant. This includes self-evaluating objects such as numbers, characters, strings, vectors, as well as all constant symbols declared by **defconstant**, such as **nil**, **t**, and ***pi*.** In addition, a list whose *car* is **quote**, such as **(quote** *foo***)**, is considered to be a constant. You can't change the value of symbol declared with **defconstant**.
If **constantp** is false of an object, then that object, considered as a form, might or might not always evaluate to the same thing.

13 Symbol Functions

The following table shows the symbol access and modification functions. The **set-***functions* (**setq** is not a **set-***function*) are functions that take two arguments, the first argument is new value and the second argument is the symbol that will be changed by this call. Those functions are particularly useful for implementing interpreters for languages embedded in Lisp. The assignment primitive may be used with **setf** and the access form. Note that there is no function to access or modify the current lexical value of a symbol.

Symbol slot	*Define form*	*Access form*	*Modification form*	*Test form*	*Unbound form*
name	-	symbol-name	-	-	-
property list	-	symbol-plist	set-symbol-plist	-	-
package	defpackage	symbol-package	set-symbol-package	-	-
function	defun	symbol-function	set-symbol-function	fboundp	fmakunbound
macro	defmacro	macro-function	set-macro-function	macro-function	fmakunbound
dynamic value	defdynamic	symbol-value	set-symbol-value	boundp	makunbound

global value	**defglobal**	**symbol-global**	**set-symbol-global**	**gboundp**	**gmakunbound**
constant value	**defconstant**	**symbol-global**	**-**	**constantp**	**gmakunbound**

(symbol-function *symbol***)** -> *<object>* **function**
(set-symbol-function *function symbol***)** -> *<object>* **function**
(setf (symbol-function *symbol***)** *function***)** -> *<object>* **function**

symbol-function returns a copy of the current global function definition named by *symbol*. An error is signaled if the symbol has no function definition; see **fboundp**. Note that the definition may be a function or may be an object representing a special form or macro. In the latter case, however, it is an error to attempt to invoke the object as a function. The corresponding assignment primitive is **set-symbol-function**; alternatively, **symbol-function** may be used with **setf**.
If it is desired to process macros, special forms, and functions equally well, as when writing an interpreter, it is best first to test the symbol with **macro-function** and **special-form-p** and then to invoke the functional value only if these two tests both yield false.

(macro-function *symbol***)** -> *<object>* **function**
(set-macro-function *function symbol***)** -> *<object>* **function**
(setf (macro-function *symbol***)** *function***)** -> *<object>* **macro**

The argument must be a symbol. If the symbol has a global definition that is a macro definition, then the expansion function (a function of two arguments, the macro-call form and an environment) is returned. If the symbol has no global function definition, or has a definition as an ordinary function or as a special form but not as a macro, then **nil** is returned. The function **macroexpand** is the best way to invoke the expansion function. The corresponding assignment primitive is **set-macro-function**; alternatively, **macro-function** may be used with **setf**.
It is possible for *both* **macro-function** and **special-form-p** to be true of a symbol. This is possible because an implementation is permitted to implement any macro also as a special form for speed.

(symbol-name *symbol***)** -> *<object>* **function**

This returns the print name of the symbol *symbol*. **OpenLisp** neutral character set is lower character.

Example:

```
(symbol-name 'xyz)       => "xyz"
(symbol-name 'Foo)       => "foo"
(symbol-name '|Foo|)     => "Foo"
```

It is an extremely bad idea to modify a string being used as the print name of a symbol. Such a modification may tremendously confuse the function **read** and the package system.

(symbol-package *symbol***)** -> *<object>* **function**
(set-symbol-package *package symbol***)** -> *<object>* **function**
(setf (symbol-package *symbol***)** *package***)** -> *<object>* **macro**

Given a symbol *sym*, **symbol-package** returns the contents of the package cell of that symbol. This will be a package object or **nil**.

(symbol-plist *symbol***)** -> *<object>* **function**
(set-symbol-plist *plist symbol***)** -> *<object>* **function**
(setf (symbol-plist *symbol***)** *plist***)** -> *<object>* **macro**

This returns the list that contains the property pairs of *symbol*; the contents of the property-list cell are extracted and returned. **setf** may be used with **symbol-plist** to destructively replace the entire property list of a symbol. This is a relatively dangerous operation, as it may destroy important information that the implementation may happen to store in property lists. Also, care must be taken that the new property list is in fact a list of even length.

(symbol-value *symbol***)** -> *<object>*	**function**
(set-symbol-value *value symbol***)** -> *<object>*	**function**
(setf (symbol-value *symbol***)** *value***)** -> *<object>*	**macro**

symbol-value returns the current value of the dynamic (special) variable named by *symbol* (introduced by **defdynamic**). An error occurs if the symbol has no dynamic value; see **boundp**. This function is particularly useful for implementing interpreters for languages embedded in Lisp. The corresponding assignment primitive is **set**; alternatively, **symbol-value** may be used with **setf**.

(symbol-global *symbol***)** -> *<object>*	**function**
(set-symbol-global *value symbol***)** -> *<object>*	**function**
(setf (symbol-global *symbol***)** *value***)** -> *<object>*	**macro**

symbol-global returns the current value of the global variable named by *symbol* (introduced by **defglobal**). An error occurs if the symbol has no global value; see **gboundp**. Note that constant symbols are really variables that cannot be changed, and so **symbol-global** may be used to get the value of a named constant. In particular, **symbol-global** of a keyword will return that keyword.
symbol-global cannot access the value of a lexical variable.
This function is particularly useful for implementing interpreters for languages embedded in Lisp. The corresponding assignment primitive is **set-symbol-global**; alternatively, **symbol-global** may be used with **setf**.

(symbol-access *symbol***)** -> *<keyword>*	**function**

Returns symbol package visibility, either **:internal** or **:external**.

(set-dynamic *value* <u>*symbol*</u>**)** -> *<object>*	**special form**
(setf (dynamic <u>*symbol*</u>**)** *value***)** -> *<object>*	**macro**

set-dynamic allows alteration of the value of a dynamic variable. **set-dynamic** causes the dynamic variable named by *symbol* to take on *value* as its value. It is an error to set the dynamic value of a symbol not declared by **defdynamic** or **dynamic-let**.

(set *symbol value***)** -> *<object>*	**function**

set allows alteration of the value of a dynamic variable. **set** causes the dynamic variable named by *symbol* to take on *value* as its value.

(synonym *symbol new-symbol***)** -> *<object>*	**function**

Creates a new symbol *new-symbol* with exactly the same properties of symbol *symbol.*

(remove-symbol *symb***)** -> *<object>*	**function**

Removes all the properties associated with the symbol *symbol* so that the next GC can collect it. This function always returns **t**.

(concat $symbol_1$ ... $symbol_N$**)** -> *<symbol>*	**function**

Creates and returns a new symbol whose print name is the concatenation of $symbol_1$ .. $symbol_N$. The new symbol is returned.

(special-form-p *symbol***)** -> *<object>*	**function**

The function **`special-form-p`** takes a symbol. If the symbol globally names a special form, then a non-**`nil`** value is returned; otherwise **`nil`** is returned. A returned non-**`nil`** value is typically a function of implementation-dependent nature that can be used to interpret (evaluate) the special form.
It is possible for *both* **`special-form-p`** and **`macro-function`** to be true of a symbol. This is possible because an implementation is permitted to implement any macro also as a special form for speed.

`(function-type` *symbol***`)`** -> *<symbol>* **`function`**

Returns a symbol which describe the type of the function of symbol *symbol* with the following values :

`subr0`	a standard library function with 0 argument.
`subr1`	a standard library function with 1 argument.
`subr2`	a standard library function with 2 arguments.
`subr3`	a standard library function with 3 arguments.
`subrn`	a standard library function with n arguments.
`subrm`	a standard library macro.
`special-form`	a pre-definied special form.
`expr`	an interpreted function.
`macro`	a macro-function.
`cpfun`	a LAP compiled function.
`cpmacro`	a LAP compiled macro-function.
`cpsubrn`	a C compiled function.
`slot-access`	a structure access function.
`<generic-function>`	a generic function.
`nil`	no function definition is associated to this symbol.

`(function-definition` *symbol***`)`** -> *<object>* **`function`**

Returns, if it exists, the function definition associated to the symbol *symbol* or **`nil`** otherwise.

`(boundp` *symbol***`)`** -> *<object>* **`function`**

`boundp` is true if the dynamic (special) variable named by *symbol* has a value; otherwise, it returns **`nil`**.

`(gboundp` *symbol***`)`** -> *<object>* **`function`**

`gboundp` is true if the global variable named by *symbol* has a value; otherwise, it returns **`nil`**.

`(fboundp` *symbol***`)`** -> *<object>* **`function`**

`fboundp` is true if the symbol has a global function definition. Note that **`fboundp`** is true when the symbol names a special form or macro. **`macro-function`** and **`special-form-p`** may be used to test for these cases.

`(makunbound` *symbol***`)`** -> *<object>* **`function`**

`makunbound` causes the dynamic variable named by *symbol* to become unbound (have no value).

Example:

```
(setf (dynamic a) 1)
(dynamic a) => 1
(makunbound 'a)
(dynamic a) => causes an error

(defun foo (x) (+ x 1))
```

```
(foo 4) => 5
(fmakunbound 'foo)
(foo 4) => causes an error
```

(gmakunbound *symbol***)** -> <*object*> **function**

gmakunbound causes the global variable named by *symbol* to become unbound (have no global value).

Example:

```
(setf a 1)
a => 1
(gmakunbound 'a)
a => causes an error
```

(fmakunbound *symbol***)** -> <*object*> **function**

fmakunbound causes the function value of function named by *symbol* to become unbound (have no function value).

Example:

```
(defun foo (x) (+ x 1))
(foo 4) => 5
(fmakunbound 'foo)
(foo 4) => causes an error
```

(macroexpand-1 *form***)** -> <*object*> **function**
(macroexpand *form***)** -> <*object*> **function**
(macroexpand-all *form***)** -> <*object*> **function**

If *form* is a macro call, then **macroexpand-1** will expand the macro call *once* and return the expansion. If *form* is not a macro call, then **nil** is returned.
A *form* is considered to be a macro call only if it is a cons whose *car* is a symbol that names a macro. Only global macro definitions (as established by **defmacro**) are considered.
macroexpand is similar to **macroexpand-1**, but repeatedly expands *form* until no more expansion can be made.
macroexpand-all is similar to **macroexpand**, but recursively expands *form* until no more expansion can be made.

14 Lists Class

ISLISP extensions to the **<list>** class.

(caar *cons***)** -> <*object*> **function**
(cadr *cons***)** -> <*object*> **function**
(cdar *cons***)** -> <*object*> **function**
(cddr *cons***)** -> <*object*> **function**
(caaar *cons***)** -> <*object*> **function**
(caadr *cons***)** -> <*object*> **function**
(cadar *cons***)** -> <*object*> **function**
(caddr *cons***)** -> <*object*> **function**
(cdaar *cons***)** -> <*object*> **function**
(cdadr *cons***)** -> <*object*> **function**
(cddar *cons***)** -> <*object*> **function**
(cdddr *cons***)** -> <*object*> **function**

(caaaar *cons***)** -> *<object>* **function**
(caaadr *cons***)** -> *<object>* **function**
(caadar *cons***)** -> *<object>* **function**
(caaddr *cons***)** -> *<object>* **function**
(cadaar *cons***)** -> *<object>* **function**
(cadadr *cons***)** -> *<object>* **function**
(caddar *cons***)** -> *<object>* **function**
(cadddr *cons***)** -> *<object>* **function**
(cdaaar *cons***)** -> *<object>* **function**
(cdaadr *cons***)** -> *<object>* **function**
(cdadar *cons***)** -> *<object>* **function**
(cdaddr *cons***)** -> *<object>* **function**
(cddaar *cons***)** -> *<object>* **function**
(cddadr *cons***)** -> *<object>* **function**
(cdddar *cons***)** -> *<object>* **function**
(cddddr *cons***)** -> *<object>* **function**

All of the compositions of up to four **car** and **cdr** operations are defined as separate functions. The names of these functions begin with **c** and end with **r**, and in between is a sequence of **a** and **d** letters corresponding to the composition performed by the function. For example:

(cddadr x) is the same as **(cdr (cdr (car (cdr x))))**

If the argument is regarded as a list, then **cadr** returns the second element of the list, **caddr** the third, and **cadddr** the fourth. If the first element of a list is a list, then **caar** is the first element of the sublist, **cdar** is the rest of that sublist, and **cadar** is the second element of the sublist, and so on.
As a matter of style, it is often preferable to define a function or macro to access part of a complicated data structure, rather than to use a long **car/cdr** string. For example, one might define a macro to extract the list of parameter variables from a lambda-expression:

```
(defmacro lambda-vars (lambda-exp) `(cadr ,lambda-exp))
```

and then use *lambda-vars* for this purpose instead of **cadr**.
Any of these functions may be used to specify a *place* for **setf**.

15 Using Lists as Sets

OpenLisp includes functions that allow a list of items to be treated as a *set*. There are functions to add, remove, and search for items in a list, based on various criteria.

(member *object list* [**:test** *test-function*]**)** -> *<boolean>* **function**
(member-if *fun list***) ->** *<boolean>* **function**
(member-if-not *fun list***) ->** *<boolean>* **function**

The *list* is searched for an element that satisfies the test. If none is found, **nil** is returned; otherwise, the tail of *list* beginning with the first element that satisfied the test is returned. The *list* is searched on the top level only. These functions are suitable for use as predicates.

Example:

```
(member 'snerd '(a b c d))                   => nil
(member-if #'numberp '(a #\Space 5.3 foo)) => (5.3 foo)
(member '(a)
   '(g (a y) (a) d (a) f) :test #'equal)   => ((a) d e (a) f)
```

(adjoin *item list* [**:test** *test-function*]**) ->** *<object>* **function**

adjoin is used to add an element to a set, provided that it is not already a member. The equality test is **eq** unless a *test-function* is provided.
(adjoin item list) == (if (member item list) list (cons item list))

(last *list***) ->** *<object>* **function**

---

last returns the last cons (*not* the last element!) of *list*. If *list* is **()**, it returns **()**.

Example:

```
(setq x '(a b c d))                  => (a b c d)
(last x)                             => (d)
(setf (cdr (last x)) '(e f))         => (e f)
x                                    => '(a b c d e f)
(last '(a b c . d))                  => (c . d)
```

(list* *arg₁* .. *argₙ***) ->** *<list>* **function**

---

list* is like **list** except that the last *cons* of the constructed list is ``dotted.'' The last argument to **list*** is used as the *cdr* of the last cons constructed; this need not be an atom. If it is not an atom, then the effect is to add several new elements to the front of a list.

Example:

(list* 'a 'b 'c 'd) => (a b c . d)
This is like
(cons 'a (cons 'b (cons 'c 'd)))
Also:
(list* 'a 'b 'c '(d e f)) => (a b c d e f)
(list* x) == x

(copy-list *list***) ->** *<list>* **function**

This returns a list that is **equal** to *list*, but not **eq**. Only the top level of list structure is copied; that is, **copy-list** copies in the *cdr* direction but not in the *car* direction. If the list is ``dotted,'' that is, **(cdr (last** *list***))** is a non-**nil** atom, this will be true of the returned list also. See also **copy-seq** and **copy-tree**.

(copy-alist *list***) ->** *<list>* **function**

---

copy-alist is for copying association lists. The top level of list structure of *list* is copied, just as for **copy-list**. In addition, each element of *list* that is a cons is replaced in the copy by a new cons with the same *car* and *cdr*.

(copy-tree *object***) ->** *<list>* **function**

---

copy-tree is for copying trees of conses. The argument *object* may be any Lisp object. If it is not a cons, it is returned; otherwise the result is a new cons of the results of calling **copy-tree** on the *car* and *cdr* of the argument. In other words, all conses in the tree are copied recursively, stopping only when non-conses are encountered. Circularities and the sharing of substructure are *not* preserved.

(endp *list***) ->** *<boolean>* **function**

The predicate **endp** is the recommended way to test for the end of a list. It is false of conses, true of **nil**, and an error for all other arguments.

(list-length *list***) ->** *<object>* **function**

---

list-length returns, as an integer, the length of *list*. **list-length** differs from **length** when the *list* is circular; **length** may fail to return, whereas **list-length** will return **nil**.

(first *list***)** -> *<object>*	**function**
(second *list***)** -> *<object>*	**function**
(third *list***)** -> *<object>*	**function**
(fourth *list***)** -> *<object>*	**function**
(fifth *list***)** -> *<object>*	**function**
(sixth *list***)** -> *<object>*	**function**
(seventh *list***)** -> *<object>*	**function**
(eighth *list***)** -> *<object>*	**function**
(ninth *list***)** -> *<object>*	**function**
(tenth *list***)** -> *<object>*	**function**

These functions are sometimes convenient for accessing particular elements of a list. **first** is the same as **car**, **second** is the same as **cadr**, **third** is the same as **caddr**, and so on except that error is not raised if list has fewer elements. Note that the ordinal numbering used here is one-origin, as opposed to the zero-origin numbering used by **nth**.

setf may be used with each of these functions to store into the indicated position of a list.

(rest *list***)** -> *<list>*	**function**

rest means the same as **cdr** but mnemonically complements **first**. **setf** may be used with **rest** to replace the *cdr* of a list with a new value. If list is **nil**, **rest** returns **nil**.

(butlast *list* **[***n***])** -> *<list>*	**function**

This creates and returns a list with the same elements as *list*, excepting the last *n* elements. *n* defaults to 1. The argument is not destroyed. If the *list* has fewer than *n* elements, then **()** is returned.

Example:

```
(butlast '(a b c d))              => (a b c)
(butlast '((a b) (c d)))          => ((a b))
(butlast '(a))                    => ()
(butlast nil)                     => ()
```

(nbutlast *list* **[***n***])** -> *<list>*	**function**

This is the destructive version of **butlast**; it changes the *cdr* of the cons *n*+1 from the end of the *list* to **nil**. *n* defaults to 1. If the *list* has fewer than *n* elements, then **nbutlast** returns **()**, and the argument is not modified. (Therefore one normally writes **(setq a (nbutlast a))** rather than simply **(nbutlast a)**.)

Example:

```
(setq foo '(a b c d))           => (a b c d)
(nbutlast foo)                  => (a b c)
foo                             => (a b c)
(nbutlast '(a))                 => ()
(nbutlast 'nil)                 => ()
```

(ldiff *list sublist***) ->** *<list>*	**function**

list should be a list, and *sublist* should be a sublist of *list*, that is, one of the conses that make up *list*. **ldiff** (meaning ``list difference'') will return a new (freshly consed) list, whose elements are those elements of *list* that appear before *sublist*. If *sublist* is not a tail of *list* (and in particular if *sublist* is **nil**), then a copy of the entire *list* is returned. The argument *list* is not destroyed.

Example:

```
(setq x '(a b c d e))              => (a b c d e)
(setq y (cdddr x))                 => (d e)
(ldiff x y)                        => (a b c)
(ldiff '(a b c d) '(c d))          => (a b c d)
```
since the sublist was not eq to any part of the list.

(nth *n list***) ->** *<object>* **function**

(nth *n list***)** returns the *n*th element of *list*, where the *car* of the list is the ``zeroth'' element. The argument *n* must be a non-negative integer. If the length of the list is not greater than *n*, then the result is **()**, that is, **nil**.

Example:

```
(nth 0 '(foo bar gack))            => foo
(nth 1 '(foo bar gack))            => bar
(nth 3 '(foo bar gack))            => ()
```

nth may be used to specify a *place* to **setf**; when **nth** is used in this way, the argument *n* must be less than the length of the *list*.
Note: that the arguments to **nth** are reversed from the order used by most other sequence selector functions such as **elt**.

(nthcdr *n list***) ->** *<list>* **function**

(nthcdr *n list***)** performs the **cdr** operation *n* times on *list*, and returns the result.

Example:

```
(nthcdr 0 '(a b c))                => (a b c)
(nthcdr 2 '(a b c))                => (c)
(nthcdr 4 '(a b c))                => ()
```

In other words, it returns the *n*th *cdr* of the list.

(rplaca *cons x***) ->** *<object>* **function**

(rplaca *cons x***)** changes the *car* of *cons* to *x* and returns (the modified) *cons*. *cons* must be a cons, but *x* may be any Lisp object.

Example:

```
(setq g '(a b c))                  => (a b c)
(rplaca (cdr g) 'd)                => (d c)
g                                  => (a d c)
```

(rplacd *cons x***) ->** *<object>* **function**

(rplacd *cons x***)** changes the *cdr* of *cons* to *x* and returns (the modified) *cons*. *cons* must be a cons, but *x* may be any Lisp object.

Example:

```
(setq g '(a b c))                    => (a b c)
(rplacd g 'd)                        => (a . d)
g                                    => (a . d)
```

(rplac *cons a d***) ->** *<object>* **function**

(rplac *cons x y***)** changes the *car* of *cons* to *x* and the *cdr* of *cons* to *y*. It returns the modified *cons*. *cons* must be a cons, but *y* may be any Lisp object.

Example:

```
(setq g '(a b c))                    => (a b c)
(rplac g 1 (list 2 3))               => (1 2 3)
g                                    => (1 2 3)
```

(displace *cons₁ cons₂***) ->** *<object>* **function**

(displace *cons₁ cons₂***)** changes the *car* of *cons₁* with the *car* of *cons₂* and the *cdr* of *cons₁* with the *cdr* of *cons₂*. It returns the modified *cons₁*.

(revappend *x y***) ->** *<object>* **function**

(revappend *x y***)** is exactly the same as **(append (reverse** *x***)** *y***)** except that it is potentially more efficient. Both *x* and *y* should be lists. The argument *x* is copied, not destroyed. Compare this with **nreconc**, which destroys its first argument.

(nconc *l₁ .. lN***) ->** *<object>* **function**

nconc takes lists as arguments. It returns a list that is the arguments concatenated together. The arguments are changed rather than copied. (Compare this with **append**, which copies arguments rather than destroying them.)

Example:

```
(setq x '(a b c))
(setq y '(d e f))
(nconc x y) => (a b c d e f)
x => (a b c d e f)
```

Note : in the example, that the value of *x* is now different, since its last cons has been **rplacd**'d to the value of *y*. If one were then to evaluate **(nconc** *x y***)** again, it would yield a piece of ``circular" list structure, whose printed representation would be **(***a b c d e f d e f d e f* **...)**, repeating forever.

Examples :

```
(nconc)                   nil     ;No side effects
(nconc nil . r)           (nconc . r)
(nconc x)                 x
(nconc x y)               (let ((p x) (q y))
                               (setf (cdr (last p)) q)
                                p)
(nconc x y . r)           (nconc (nconc x y) . r)
```

(nconc1 *list x***) ->** *<object>* **function**

nconc1 takes a list *list* and an atom *x*. It returns a list that is the list *list* concatenated with **(list** *x***)**.

(nreconc *x y***) ->** *<object>* **function**

(nreconc *x y***)** is exactly the same as **(nconc (nreverse** *x***)** *y***)** except that it is potentially more efficient. Both *x* and *y* should be lists. The argument *x* is destroyed. Compare this with **revappend**.

Example:

```
(setq planets '(jupiter mars earth venus mercury))
(setq more-planets '(saturn uranus pluto neptune))
(nreconc more-planets planets)
=> (neptune pluto uranus saturn jupiter mars earth venus mercury)
  and now the value of more-planets is not well defined
```

(cirlist l_1 .. l_N**) ->** *<object>* **function**

cirlist takes a list of arguments. It returns a circurlar list made of those arguments.

Example:

```
(setq x '(a b c))
(cirlist x 1 2) => ((a b c) 1 2 (a b c) 1 2 (a b c) 1 2 …)
x => (a b c)
```

(subst *new old tree***) ->** *<object>* **function**

(subst *new old tree***)** makes a copy of *tree*, substituting *new* for every subtree or leaf of *tree* (whether the subtree or leaf is a *car* or a *cdr* of its parent) such that *old* and the subtree or leaf satisfy the test. It returns the modified copy of *tree*. The original *tree* is unchanged, but the result tree may share with parts of the argument *tree*.

Example:

```
(subst 'tempest 'hurricane
       '(shakespeare wrote (the hurricane)))
   => (shakespeare wrote (the tempest))

(subst 'foo 'nil '(shakespeare wrote (twelfth night)))
   => (shakespeare wrote (twelfth night . foo) . foo)

(subst '(a . cons) '(old . pair)
       '((old . spice) ((old . shoes) old . pair) (old . pair))
       :test #'equal)
   => ((old . spice) ((old . shoes) a . cons) (a . cons))
```

(nsubst *new old tree***) ->** *<object>* **function**

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

16 Logical Operations on Numbers

The logical operations in this section require integers as arguments; it is an error to supply a non-integer as an argument. The functions all treat integers as if they were represented in two's-complement notation.
The logical operations provide a convenient way to represent an infinite vector of bits. Let such a conceptual vector be indexed by the non-negative integers. Then bit *j* is assigned a ``weight''. Assume that only a finite number of bits are 1's or only a finite number of bits are 0's. A vector with only a finite number of one-bits is

represented as the sum of the weights of the one-bits, a positive integer. A vector with only a finite number of zero-bits is represented as **-1** minus the sum of the weights of the zero-bits, a negative integer.
This method of using integers to represent bit-vectors can in turn be used to represent sets. Suppose that some (possibly countably infinite) universe of discourse for sets is mapped into the non-negative integers. Then a set can be represented as a bit vector; an element is in the set if the bit whose index corresponds to that element is a one-bit. In this way all finite sets can be represented (by positive integers), as well as all sets whose complements are finite (by negative integers). The functions **logior**, **logand**, and **logxor** defined below then compute the union, intersection, and symmetric difference operations on sets represented in this way.

(logior *n_1 .. n_N***)** -> *<integer>* **function**

This returns the bit-wise logical *inclusive or* of its arguments. If no argument is given, then the result is zero, which is an identity for this operation.

(logand *n_1 .. n_N***)** -> *<integer>* **function]**

This returns the bit-wise logical *and* of its arguments. If no argument is given, then the result is `-1`, which is an identity for this operation.

(logandc1 *n_1 n_2***)** -> *<integer>* **function]**
(logandc2 *n_1. n_2***)** -> *<integer>* **function]**
(logorc2 *n_1 n_2***)** -> *<integer>* **function]**
(logorc2 *n_1 n_2***)** -> *<integer>* **function]**

Return the following equivances:

```
(logandc1 n1 n2) == (logand (lognot n1) n2)
(logandc2 n1 n2) == (logand n1 (lognot n2))
(logorc1 n1 n2)  == (logior (lognot n1) n2)
(logorc2 n1 n2)  == (logior n1 (lognot n2))
```

(logxor *n_1 .. n_N***)** -> *<integer>* **function]**

This returns the bit-wise logical *exclusive or* of its arguments. If no argument is given, then the result is zero, which is an identity for this operation.

(logeqv *n_1 .. n_N***)** -> *<integer>* **function**

This returns the bit-wise logical *equivalence* (also known as *exclusive nor*) of its arguments. If no argument is given, then the result is `-1`, which is an identity for this operation.

(lognot *n***)** -> *<integer>* **function**

This returns the bit-wise logical *not* of its argument. Every bit of the result is the complement of the corresponding bit in the argument.

```
(logbitp j (lognot x)) == (not (logbitp j x))
```

(lognand *n_1 n_2***)** -> *<integer>* **function**
(lognor *n_1 n_2***)** -> *<integer>* **function**

These are the other two non-trivial bit-wise logical operations on two arguments. Because they are not associative, they take exactly two arguments rather than any non-negative number of arguments.

```
(lognand n1 n2) == (lognot (logand n1 n2))
(lognor n1 n2) == (lognot (logior n1 n2))
```

(logtest *n_1 n_2***)** -> *<integer>* **function**

logtest is a predicate that is true if any of the bits designated by the 1's in n_1 are 1's in n_2.
(logtest x y) == (not (zerop (logand x y)))

(logbitp *integer index***)** -> <*integer*> **function**

---

logbitp is true if the bit in *integer* whose index is *index* (that is, its weight is
(logbitp 2 6) is true
(logbitp 0 6) is false
(logbitp k n) == (ldb-test (byte 1 k) n)

(logcount *n***)** -> <*integer*> **function**

This function returns the number of bits in *n*. The result is always a non-negative integer.

(ash *integer count***)** -> <*integer*> **function**

This function shifts *integer* arithmetically left by *count* bit positions if *count* is positive, or right by ***-count*** bit positions if *count* is negative. The sign of the result is always the same as the sign of *integer*.

(1+ *number***)** -> <*integer*> **function**
(1- *number***)** -> <*integer*> **function**

(1+ *x***)** is the same as **(+** *x* **1)**.
(1- *x***)** is the same as **(-** *x* **1)**. Note that the short name may be confusing: **(1-** *x***)** does *not* mean 1-*x*; rather, it means *x*-1.

(rem n_1 n_2**)** -> <*integer*> **function**

---

rem performs the remainder of two integer arguments.

Example:

```
(rem 13 4)                  => 1
(rem -13 4)                 => -1
(rem 13 -4)                 => 1
(rem -13 -4)                => -1
(rem 13.4 1)                => 0.4
(rem -13.4 1)               => -0.4
```

(random *n***)** -> <*number*> **function**

(random *n***)** accepts a positive number *n* and returns a number of the same kind between zero (inclusive) and *n* (exclusive). The number *n* may be an integer or a floating-point number. An approximately uniform choice distribution is used. If *n* is an integer, each of the possible results occurs with (approximate) probability 1/*n*.

(set-random *n***)** -> <*integer*> **function**

---

set-random changes the state of internal random routines using non-negative integer *n* to generate the next random number. You can typically use the clock value for that purpose.

(integer-length *n***)** -> <*integer*> **function**

This function is useful in two different ways. First, if *n* is non-negative, then its value can be represented in unsigned binary form in a field whose width in bits is no smaller than **(integer-length *n*)**. Second, regardless of the sign of *n*, its value can be represented in signed binary two's-complement form in a field whose width in bits is no smaller than **(+ (integer-length *n*) 1)**.

Example:

```
(integer-length 0)                    => 0
(integer-length 1)                    => 1
(integer-length 3)                    => 2
(integer-length 4)                    => 3
(integer-length 7)                    => 3
(integer-length -1)                   => 0
(integer-length -4)                   => 2
(integer-length -7)                   => 3
(integer-length -8)                   => 3
```

17 Predicates on Numbers

"*most-positive-unboxed* -> *<integer>* **constant**

The maximal unboxed positive integer. The value depends on architecture.

"*most-negative-unboxed* -> *<integer>* **constant**

The maximal unboxed negative integer. The value depends on architecture.

"*most-positive-boxed* -> *<integer>* **constant**

The maximal boxed positive integer. The value depends on architecture.

"*most-negative-boxed* -> *<integer>* **constant**

The maximal boxed negative integer. The value depends on architecture.

(zerop *n***)** -> *<boolean>* **function**

This predicate is true if *number* is zero (the integer zero, a floating-point zero, or a complex zero), and is false otherwise. Regardless of whether an implementation provides distinct representations for positive and negative floating-point zeros, **(zerop -0.0)** is always true. It is an error if the argument *number* is not a number.

(plusp *n***)** -> *<boolean>* **function**

This predicate is true if *number* is strictly greater than zero, and is false otherwise. It is an error if the argument *number* is not a non-complex number.

(minusp *n***)** -> *<boolean>* **function**

This predicate is true if *number* is strictly less than zero, and is false otherwise. Regardless of whether an implementation provides distinct representations for positive and negative floating-point zeros, **(minusp -0.0)** is always false.

(evenp *n***)** -> *<boolean>* **function**

This predicate is true if *n* is even (divisible by 2), and false otherwise.

(oddp *n***)** -> *<boolean>* **function**

This predicate is true if *n* is odd (not divisible by 2), and false otherwise.

(bignump *n***)** -> *<boolean>* **function**

This predicate is true if *number* is bignum.

(rationalp *n***)** -> *<boolean>* **function**

This predicate is true if *number* is a rational number.

18 Other predicate

(atom *o***)** -> *<boolean>* **function**

The predicate **atom** is true if its argument is not a cons, and otherwise is false. Note that **(atom '())** is true, because **() == nil**.

(type-of *object***)** -> *<class>* **function**

(type-of *object***)** returns class of which the *object* is a member. If the argument is a user-defined named structure created by **defclass** or **defstruct**, then **type-of** will return the type name of that structure. **type-of** is an alias of **class-of** ISLISP special form (see its definition in ISLISP document).

(externalp *object***)** -> *<boolean>* **function**

Returns **t** if *object* is an external pointer (*i.e.* a C ou C++ pointer) or **nil** otherwise.

(conditionp *object***)** -> *<boolean>* **function**

conditionp is true if its argument is a condition, and otherwise is false.

19 String Construction and Manipulation

A string is a specialized vector (one-dimensional array) whose elements are characters. Any string-specific function defined in this chapter whose name begins with the prefix **string** will accept a symbol instead of a string as an argument *provided* that the operation never modifies that argument; the print name of the symbol is used. In this respect the string-specific sequence operations are not simply specializations of generic versions; the generic sequence operations never accept symbols as sequences. This slight inelegance is permitted in the name of pragmatic utility. One may get the effect of having a generic sequence function operate on either symbols or strings by applying the coercion function **string** to any argument whose data type is in doubt.

(string-equal str_1 str_2**)** -> *<boolean>* **function**
(string-lessp str_1 str_2**)** -> *<boolean>* **function**
(string-greaterp str_1 str_2**)** -> *<boolean>* **function**
(string-not-greaterp str_1 str_2**)** -> *<boolean>* **function**
(string-not-lessp str_1 str_2**)** -> *<boolean>* **function**
(string-not-equal str_1 str_2**)** -> *<boolean>* **function**

These are exactly like **string=**, **string<**, **string>**, **string<=**, **string>=**, and **string/=**, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if **char-lessp** were used instead of **char<** for comparing characters.

(char *string index***)** -> *<character>* **function**

The given *index* must be a non-negative integer less than the length of *string*, which must be a string. The character at position *index* of the string is returned as a character object.

(set-char *char string index***)** -> *<character>*	**function**
(setf (char *string index***)** *char***)** -> *<character>*	**special operator**

The given *index* must be a non-negative integer less than the length of *string*, which must be a string and *char* which must be a character. The character at position *index* of the string is modified by character object.

(string-downcase *string***)** -> *<object>*	**function**
(string-upcase *string***)** -> *<string>*	**function**
(string-capitalize *string***)** -> *<string>*	**function**

string-upcase returns a string just like *string* with all lowercase characters replaced by the corresponding uppercase characters. More precisely, each character of the result string is produced by applying the function **char-upcase** to the corresponding character of *string*.
string-downcase is similar, except that uppercase characters are converted to lowercase characters (using **char-downcase**). The argument is not destroyed. However, if no characters in the argument require conversion, the result may be either the argument or a copy of it, at the implementation's discretion. **string-capitalize** produces a copy of string such that, for every word in the copy, the first character of the word, if case-modifiable, is uppercase and any other case-modifiable characters in the word are lowercase. For the purposes of **string-capitalize**, a word is defined to be a consecutive subsequence consisting of alphanumeric characters or digits, delimited at each end either by a non-alphanumeric character or by an end of the string.

Example:

```
(string-upcase "Dr. Livingstone, I presume?")
   => "DR. LIVINGSTONE, I PRESUME?"
(string-downcase "Dr. Livingstone, I presume?")
   => "dr. livingstone, i presume?"
(string-upcase "Dr. Livingstone, I presume?" :start 6 :end 10)
   => "Dr. LiVINGstone, I presume?"
(string-capitalize " hello ")
   => " Hello "
```

(nstring-downcase *string***)** -> *<string>*	**function**
(nstring-upcase *string***)** -> *<string>*	**function**
(nstring-capitalize *string***)** -> *<string>*	**function**

These three functions are just like **string-upcase**, **string-downcase** and **string-captitalize** but destructively modify the argument *string* by altering case-modifiable characters as necessary.

(string-trim *character-bag string***)** -> *<string>*	**function**
(string-left-trim *character-bag string***)** -> *<string>*	**function**
(string-right-trim *character-bag string***)** -> *<string>*	**function**

string-trim returns a substring of *string*, with all characters in *character-bag* stripped off the beginning and end. The function **string-left-trim** is similar but strips characters off only the beginning; **string-right-trim** strips off only the end. The argument character-bag may be any sequence containing characters.

Example:

```
(string-trim '(#\Space #\Tab #\Newline) " garbanzo beans
        ") => "garbanzo beans"
(string-trim " (*)" " ( *three (silly) words* ) ")
   => "three (silly) words"
```

```
(string-left-trim " (*)" " ( *three (silly) words* ) ")
   => "three (silly) words* ) "
(string-right-trim " (*)" " ( *three (silly) words* ) ")
   => " ( *three (silly) words"
```

If no characters need to be trimmed from the string, then the argument string itself is returned.

(string-split *character-bag string* **[***keep***])** -> *<list>* **function**

---

string-split returns a list of strings using all characters in *character-bag* as word delimiter for string *string*. The argument *character-bag* may be any sequence containing characters. If the optional argument *keep* is non-**nil**, blank strings are preserved.

Example:

```
(string-split '(#\Space #\.) " garbanzo beans.")
   => ("garbanzo" "beans")
(string-split "." "computer.eligis.com")
   ⇨ ("computer" "eligis" "com")
(string-split ";" "1;;3")
   => ("1" "3")
(string-split ";" "1;;3" t)
   => ("1" "" "3")
```

(string-replace *regex to string***)** -> *<string>* **function**

---

string-replace replaces each substring of *string* that matches *regex* with *to*.

Example:

```
(string-replace "f[o]+" "bar" "A string with foo")
   => "A string with bar"
```

20 Vector Class Functions

(svref *vector index***)** -> *<object>* **function**

The first argument must be a simple general vector, that is, an object of type **<general-vector>**. The element of the *vector* specified by the integer *index* is returned. The *index* must be non-negative and less than the length of the vector. **setf** may be used with **svref** to destructively replace a simple-vector element with a new value. **svref** is identical to **aref** except that it requires its first argument to be a simple vector. **svref** may be faster than **aref** in situations where it is applicable.

(svset *vector item object***)** -> *<object>* **function**

---

svset is identical to **set-aref** except that it requires its first argument to be a simple vector. **svset** may be faster than **set-aref** in situations where it is applicable.

(vector-type *vector* **[***type***])** -> *<symbol>* **function**
(setf (vector-type *vector***)** *type***)** -> *<symbol>* **function**

---

vector-type is internally used to tag standard vectors. This function is used the object system and structure packages.

21 Bit Vector Functions

A bit vector (or *generalized boolean*) is a special vector type that contains only **1** or **0**. It is internally optimized to use less space and is potentially faster. A bit vector is of type **<simple-bit-vector>** which is a direct subtype of **<general-vector>**. The complete class precedence list is hence: **<object>**, **<basic-array>**, **<basic-array*>**, **<general-array*>**, **<basic-vector>**, **<general-vector>**, **<simple-bit-vector>**. As such, all operations valid for a **<general-vector>**, especially sequence operations, are also valid for a **<simple-bit-vector>**.

The **#[n]*bb** ... **bb** syntax represents a **<simple-bit-vector>** where *n* is the vector dimension and **b** is either **1** or **0**.

#*	bit vector of 0 elements ()
#*1001	bit vector of 4 elements (1, 0, 0, 1)
#8*	bit vector of 8 elements (0, 0, 0, 0, 0, 0, 0, 0)
#8*10010	bit vector of 8 elements (1, 0, 0, 1, 0, 0, 0, 0)
#8*10011	bit vector of 8 elements (1, 0, 0, 1, 1, 1, 1, 1)

(create-simple-bit-vector *n [init]***)** -> *<simple-bit-vector>* **function**
(make-instance '<simple-bit-vector> *n [init]***)** -> *<simple-bit-vector>* **function**

Create a **<simple-bit-vector>** of n elements initialized to init (either **1** or **0**). If init is not supplied, **0** is used.

Example:

```
(create-simple-bit-vector 8)        => #*00000000
(create-simple-bit-vector 8 0)      => #*00000000
(create-simple-bit-vector 8 1)      => #*11111111
(create-simple-bit-vector 8 t)      => error!
```

(create-simple-bit-vector-p *object***)** -> *<boolean>* **function**

Return **t** if object is of type **<simple-bit-vector>** , **nil** otherwise.

(bit *vector index***)** -> *<integer>* **function**

The first argument must be a simple bit vector, that is, an object of type **<simple-bit-vector>**. The element of the *vector* specified by the integer *index* is returned. The *index* must be non-negative and less than the length of the vector. **setf** may be used with **bit** to destructively replace a simple-vector element with a new value.

Example:

```
(defglobal x #8*)        => #*00000000
(bit x 2)                => 0
(setf (bit x 2) 1)       => 1
(bit x 2)                => 1
x                        => #*00100000
```

(set-bit *object vector item***)** -> *<integer>* **function**
(setf (bit *vector item***)** *object***)** -> *<integer>* **function**

set-bit is identical to **set-elt** except that it requires its *vector* argument to be a simple bit vector. **set-bit** may be faster than **set-elt** in situations where it is applicable. The *object* is either 1 or 0

(bit-and *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**

(bit-andc1 *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-andc2 *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-eqv *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-ior *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-nand *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-nor *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-orc1 *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-orc2 *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-xor *bv1 bv2* **[***opt-arg***])** -> *<simple-bit-vector>* **function**
(bit-not *bv* **[***opt-arg***])** -> *<simple-bit-vector>* **function**

These functions perform bit-wise logical operations on *bv1* and *bv2* and return a **<simple-bit-vector>** such that any given bit of the result is produced by operating on corresponding bits from each of the arguments. In the case of **bit-not**, a **<simple-bit-vector>** is returned that contains a copy of *bv* with all the bits inverted. If *opt-arg* is of type **<simple-bit-vector>**) the contents of the result are destructively placed into *opt-arg*. If *opt-arg* is the symbol **t**, *bv* or *bv1* is replaced with the result; if *opt-arg* is **nil** or omitted, a **<simple-bit-vector>** is created to contain the result.

22 Character Class Functions

(char-equal c_1 c_2**)** -> *<boolean>* **function**
(char-not-equal c_1 c_2**)** -> *<boolean>* **function**
(char-lessp c_1 c_2**)** -> *<boolean>* **function**
(char-not-lessp c_1 c_2**)** -> *<boolean>* **function**
(char-greaterp c_1 c_2**)** -> *<boolean>* **function**
(char-not-greaterp c_1 c_2**)** -> *<boolean>* **function**

The predicate **char-equal** is like **char=**, and similarly for the others, except according to a different ordering such that differences of bits attributes and case are ignored.

For the standard characters, the ordering is such that **A=a**, **B=b**, and so on, up to **Z=z**, and furthermore either **9<A** or **Z<0**.

Example:

```
(char-equal #\A #\a)    => t
(char= #\A #\a)   => nil
(char-equal #\A #\Control-A)  => nil ;; strange, but this is true in Common Lisp
```

(upper-case-p *char***)** -> *<boolean>* **function**
(lower-case-p *char***)** -> *<boolean>* **function**
(both-case-p *char***)** -> *<boolean>* **function**

The argument *char* must be a character object.
upper-case-p is true if the argument is an uppercase character, and otherwise is false.
lower-case-p is true if the argument is a lowercase character, and otherwise is false.
both-case-p is true if the argument is an uppercase character and there is a corresponding lowercase character (which can be obtained using **char-downcase**), or if the argument is a lowercase character and there is a corresponding uppercase character (which can be obtained using **char-upcase**).
If a character is either uppercase or lowercase, it is necessarily alphabetic (and therefore is graphic, and therefore has a zero bits attribute). Of the standard characters (as defined by **standard-char-p**), the letters **A** through **Z** are uppercase and **a** through **z** are lowercase.

(standard-char-p *char***)** -> *<boolean>* **function**

The argument *char* must be a character object. **standard-char-p** is true if the argument is an ASCII character (in the range 0x00 – 0x7F).

Example:

```
(standard-char-p #\a)   => t
(standard-char-p #\é)   => nil
```

(graphic-char-p *char***)** -> <*boolean*> **function**

The argument *char* must be a character object. **graphic-char-p** is true if the argument is a "graphic" (printing) character, and false if it is a "non-graphic" (formatting or control) character. Graphic characters have a standard textual representation as a single glyph, such as **A** or ***** or **=**. By convention, the space character is considered to be graphic.

(alpha-char-p *char***)** -> <*boolean*> **function**

The argument *char* must be a character object. **alpha-char-p** is true if the argument is an alphabetic character, and otherwise is false. Of the standard characters (as defined by **standard-char-p**), the letters **A** through **Z** and **a** through **z** are alphabetic.

(alphanumericp *char***)** -> < *boolean*> **function**

The argument *char* must be a character object. **alphanumericp** is true if *char* is either alphabetic or numeric. By definition,
(alphanumericp x) == (or (alpha-char-p x) (not (null (digit-char-p x))))
Of the standard characters (as defined by **standard-char-p**), the characters **0** through **9**, **A** through **Z**, and **a** through **z** are alphanumeric.

(digit-char-p *char* **[***radix***])** -> <*integer*> **function**

The argument *char* must be a character object, and *radix* (default value 10) must be a non-negative integer. If *char* is not a digit then **digit-char-p** is false; otherwise it returns a non-negative integer that is the ``weight'' of *char* in that radix. Of the standard characters, the characters **0** through **9** are digits. The weights of **0** through **9** are the integers 0 through 9, and of **A** through **Z** (and also **a** through **z**) are 10 through 35. **digit-char-p** returns the weight for one of these digits if and only if its weight is strictly less than *radix*.

(char-upcase *char***)** -> < *character*> **function**
(char-downcase *char***)** -> < *character*> **function**

The argument *char* must be a character object. **char-upcase** attempts to convert its argument to an uppercase equivalent; **char-downcase** attempts to convert its argument to a lowercase equivalent.

(char-int *char***)** -> <*integer*> **function**

The argument *char* must be a character object. **char-int** returns a non-negative integer encoding the character object.
(char= c1 c2) == (= (char-int c1) (char-int c2))
for characters c_1 and c_2.

(int-char *c***)** -> <*character*> **function**

The argument must be a non-negative integer. **int-char** returns a character object *c* such that **(char-int** *c***)** is equal to *integer*, if possible; otherwise **int-char** returns false.

23 Sequence Class Functions

Some of sequence functions accept an optional test function. If this test function is not provided **eq** is used.

(make-sequence *sequence-type size* [*initial-element*]**)** -> *<sequence>* **function**

This returns a sequence of type *type* and of length *size*, each of whose elements has been initialized to the *initial-element* argument. If specified, the *initial-element* argument must be an object that can be an element of a sequence of type *type*.

Example:

```
(make-sequence '<string> 10 #\A)        -> "AAAAAAAAAA"
(make-sequence '<list> 5)               -> (nil nil nil nil nil)
(make-sequence '<list> 5 2)             -> (2 2 2 2 2)
```

(count *item sequence* [**:test** *test-function*]**)** -> *<integer>* **function**
(count-if *predicate sequence***)** -> *<integer>* **function**
(count-if-not *predicate sequence***)** -> *<integer>* **function**

The result is always a non-negative integer, the number of elements in the specified subsequence of *sequence* satisfying the test.

Example:

```
(count-if #'zerop #(2 0 3 0 4))              -> 2
(count-if-not #'lower-case-p "Some String")  -> 2
```

(copy-seq *sequence***)** -> *<sequence>* **function**

A copy is made of the argument *sequence*; the result is **equal** to the argument but not **eq** to it.
(copy-seq x) == (subseq x 0)
but the name **copy-seq** is more perspicuous when applicable.

(reduce *function sequence***)** -> *<object>* **function**

The **reduce** function combines all the elements of a sequence using a binary operation; for example, using + one can add up all the elements.
The specified subsequence of the sequence is combined or “reduced” using the function, which must accept two arguments. The reduction is left-associative.

Example:

```
(reduce #'+ '(1 2 3 4))                 -> 10
(reduce #'list '(1 2 3 4))              -> (((1 2) 3) 4)
```

(concatenate *result-type* seq_1 ... seq_N**)** -> *<object>* **function**

The result is a new sequence that contains all the elements of all the sequences in order. All of the sequences are copied from; the result does not share any structure with any of the argument sequences (in this **concatenate** differs from **append**). The type of the result is specified by *result-type*, which must be a subtype of **<sequence>**, as for the function coerce. It must be possible for every element of the argument sequences to be an element of a sequence of type result-type.

If only one sequence argument is provided and it has the type specified by *result-type*, **con**

is required to copy the argument rather than simply returning it. If a copy is not required, but only possibly type conversion, then the **convert** special form may be appropriate.

(map *result-type function seq_1 ... seq_N***)** -> *<object>* **function**

The *function* must take as many arguments as there are sequences provided; at least one sequence must be provided. The result of map is a sequence such that element *j* is the result of applying function to element *j* of each of the argument sequences. The result sequence is as long as the shortest of the input sequences.

If the *function* has side effects, it can count on being called first on all the elements numbered 0, then on all those numbered 1, and so on.

The type of the result sequence is specified by the argument *result-type* (which must be a subtype of the type **<sequence>**). In addition, one may specify **nil** for the result type, meaning that no result sequence is to be produced; in this case the function is invoked only for effect, and **map** returns **nil**. This gives an effect similar to that of **mapc**.

(some *predicate seq_1 ... seq_N***)** -> *<object>* **function**

some returns as soon as any invocation of *predicate* returns a non-**nil** value; **some** returns that value. If the end of a sequence is reached, some returns **nil**. Thus, considered as a predicate, it is true if *some* invocation of *predicate* is true.

(every *predicate seq_1 ..seq_N***)** -> *<object>* **function**

every returns **nil** as soon as any invocation of *predicate* returns **nil**. If the end of a sequence is reached, **every** returns a non-**nil** value. Thus, considered as a predicate, it is true if *every* invocation of *predicate* is true.

(notany *predicate seq_1 ..seq_N***)** -> *<object>* **function**

notany returns **nil** as soon as any invocation of *predicate* returns a non-**nil** value. If the end of a sequence is reached, **notany** returns a non-**nil** value. Thus, considered as a predicate, it is true if *no* invocation of *predicate* is true.

(notevery *predicate seq_1 .. seq_N***)** -> *<object>* **function**

notevery returns a non-**nil** value as soon as any invocation of *predicate* returns **nil**. If the end of a sequence is reached, **notevery** returns **nil**. Thus, considered as a predicate, it is true if *not every* invocation of *predicate* is true.

(find *item sequence* [**:test** *test-function*]**)** -> *<object>* **function**
(find-if *predicate sequence***)** -> *<object>* **function**
(find-if-not *predicate sequence***)** -> *<object>* **function**

If the *sequence* contains an element satisfying the test, then the leftmost such element is returned; otherwise **nil** is returned.

(position *item sequence* [**:test** *test-function*]**)** -> *<integer>* **function**
(position-if *predicate sequence***)** -> *<integer>* **function**
(position-if-not *predicate sequence***)** -> *<integer>* **function**

If the *sequence* contains an element satisfying the test, then the index within the sequence of the leftmost such element is returned as a non-negative integer; otherwise **nil** is returned.

(substitute *newitem olditem sequence* [**:test** *test-function*]**) ->** *<object>* **function**
(substitute-if *newitem predicate sequence***) ->** *<object>* **function**
(substitute-if-not *newitem predicate sequence***) ->** *<object>* **function**

The result is a sequence of the same kind as the argument *sequence* that has the same elements except that those satisfying the test (see above) have been replaced by *newitem*. This is a non-destructive operation; the result is a copy of the input sequence, save that some elements are changed.

Example:

```
(substitute 1 2 '(1 2 3 4))              => (1 1 3 4)
(substitute 1 2 '())                     => ()
(substitute 'foo 2 '(1 2 3 4))           => (1 foo 3 4)
(substitute 1 2 #(1 2 3 4))              => #(1 1 3 4)
(substitute 1 2 #())                     => #()
(substitute #\1 #\2 "1234")              => "1134"
(substitute #\1 #\2 "")                  => ""
```

(nsubstitute *newitem olditem sequence* [**:test** *test-function*]**) ->** *<object>* **function**
(nsubstitute-if *newitem predicate sequence***) ->** *<object>* **function**
(nsubstitute-if-not *newitem predicate sequence***) ->** *<object>* **function**

This is the destructive counterpart to `substitute`. The result is a sequence of the same kind as the argument *sequence* that has the same elements except that those satisfying the test (see above) have been replaced by *newitem*. This is a destructive operation. The argument *sequence* may be destroyed and used to construct the result; however, the result may or may not be `eq` to *sequence*.

(remove *object sequence* [**:test** *test-function*]**) ->** *<object>* **function**
(remove-if *predicate sequence***) ->** *<object>* **function**
(remove-if-not *predicate sequence***) ->** *<object>* **function**

The result is a sequence of the same kind as the argument *sequence* that has the same elements except that those satisfying the test function (default is **eq**) have been removed. This is a non-destructive operation; the result is a copy of the input *sequence*, save that some elements are not copied. Elements not removed occur in the same order in the result as they did in the argument.

(delete *object list* [**:test** *test-function*]**) ->** *<object>* **function**
(delete-if *predicate sequence***) ->** *<object>* **function**
(delete-if-not *predicate sequence***) ->** *<object>* **function**

This is the destructive counterpart to **remove**. The result is a sequence of the same kind as the argument *sequence* that has the same elements except that those satisfying the test function (default is **eq**) have been deleted. This is a destructive operation. The argument *sequence* may be destroyed and used to construct the result; however, the result may or may not be **eq** to *sequence*. Elements not deleted occur in the same order in the result as they did in the argument.

(remove-duplicates *sequence***) ->** *<object>* **function**
(delete-duplicates *sequence***) ->** *<object>* **function**

The elements of *sequence* are compared pairwise, and if any two match, then the one occurring earlier in the sequence is discarded. The result is a sequence of the same kind as the argument *sequence* with enough elements removed so that no two of the remaining elements match. The order of the elements remaining in the result is the same as the order in which they appear in sequence.
remove-duplicates is the non-destructive version of this operation. The result of **remove-duplicates** may share with the argument *sequence*; a list result may share a tail with an input list, and the result may be **eq** to the input sequence if no elements need to be removed.
delete-duplicates may destroy the argument *sequence.*

(intersection *list-1 list-2* [**:test** *test-function*]**) ->** *<object>* **function**

(intersection *list-1 list-2* [**:test** *test-function*]**)** **->** *<object>* **function**

The **intersection** and **nintersection** functions return a list that contains every element that occurs in both *list-1* and *list-2*. **nintersection** is the destructive version of **intersection**. It performs the same operation, but may destroy *list-1* using its cells to construct the result. *list-2* is not destroyed. The **intersection** operation is described as follows. For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, **:test** is used to determine whether it satisfies the test. The first argument to the **:test** function is an element of *list-1*; the second argument is an element of *list-2*. If **:test** is not supplied, **eql** is used.

(union *list-1 list-2* [**:test** *test-function*]**)** **->** *<object>* **function**
(union *list-1 list-2* [**:test** *test-function*]**)** **->** *<object>* **function**

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

For all possible ordered pairs consisting of one element from *list-1* and one element from *list-2*, **:test** is used to determine whether they satisfy the test. The first argument to the **:test** function is the part of the element of *list-*; the second argument is the part of the element of list-2.

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

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

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

(search *sequence1 sequence2* [**:test** *test-function*]**)** -> *<integer>* **function**

A search is conducted for a subsequence of *sequence2* that element-wise matches *sequence1* (using **eql**). If there is no such subsequence, the result is **nil**; if there is, the result is the index into *sequence2* of the leftmost element of the leftmost such matching subsequence.

(mismatch *sequence1 sequence2* [**:test** *test-function*]**)** -> *<integer>* **function**

The specified subsequences of *sequence1* and *sequence2* are compared element-wise. If they are of equal length and match in every element, the result is **nil**. Otherwise, the result is a non-negative integer. This result is the index within *sequence1* of the leftmost position at which the two subsequences fail to match; or, if one subsequence is shorter than a matching prefix of the other, the result is the index relative to *sequence1* beyond the last position tested.

24 A-List Functions

An *association list*, or *a-list*, is a data structure used very frequently in Lisp. An *a-list* is a list of pairs (conses); each pair is an association. The *car* of a pair is called the *key*, and the *cdr* is called the *datum*.
An advantage of the *a-list* representation is that an *a-list* can be incrementally augmented simply by adding new entries to the front. Moreover, because the searching function **assoc** searches the *a-list* in order, new entries can ``shadow'' old entries. If an *a-list* is viewed as a mapping from keys to data, then the mapping can be not only augmented but also altered in a non-destructive manner by adding new entries to the front of the *a-list*. Sometimes an *a-list* represents a bijective mapping, and it is desirable to retrieve a key given a datum. It is permissible to let **nil** be an element of an *a-list* in place of a pair. Such an element is not considered to be a pair but is simply passed over when the a-list is searched by **assoc**.

(acons *key datum a-list***)** -> *<list>* **function**

acons constructs a new association list by adding the pair **(***key . datum***)** to the old *a-list*.
(acons x y a) == (cons (cons x y) a)

(pairlis *keys data* **[** *a-list* **])** -> *<list>* **function**

pairlis takes two lists and makes an association list that associates elements of the first list to corresponding elements of the second list. It is an error if the two lists *keys* and *data* are not of the same length. If the optional argument *a-list* is provided, then the new pairs are added to the front of it.
The new pairs may appear in the resulting a-list in any order; in particular, either forward or backward order is permitted. Therefore the result of the call

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

might be

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

but could equally well be

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

(sublis *a-list s***)** -> *<list>* **function**

sublis makes substitutions for objects in a tree (a structure of conses). The first argument to **sublis** is an association list. The second argument is the tree in which substitutions are to be made, as for **subst**. **sublis** looks at all subtrees and leaves of the tree; if a subtree or leaf appears as a key in the association list (that is, the key and the subtree or leaf satisfy the test), it is replaced by the object with which it is associated. This operation is non-destructive. In effect, **sublis** can perform several **subst** operations simultaneously.

Example:

```
(sublis '((x . 100) (z . zprime))
        '(plus x (minus g z x p) 4 . x))
   => (plus 100 (minus g zprime 100 p) 4 . 100)
```

(assoc *item a-list***)** -> *<list>* **function**
(assoc-if *fn a-list***)** -> *<list>* **function**
(assoc-if-not *fn a-list***)** -> *<list>* **function**

Each of these searches the association list *a-list*. The value is the first pair in the a-list such that the *car* of the pair satisfies the test, or **nil** if there is no such pair in the *a-list*.

(cassoc *item a-list***)** -> *<list>* **function**

cassoc returns the value associated to *item* in *a-list* or **nil** is no association exists.

(rassoc *item a-list***)** -> *<list>* **function**
(rassoc-if *fn a-list***)** -> *<list>* **function**
(rassoc-if-not *fn a-list***)** -> *<list>* **function**

rassoc is the reverse form of **assoc**; it searches for a pair whose **cdr** satisfies the test, rather than the **car**. If the a-list is considered to be a mapping, then **rassoc** treats the a-list as representing the inverse mapping. For example:

```
(rassoc 'a '((a . b) (b . c) (c . a) (z . a))) => (c . a)*
```

25 Rational Functions

OpenLisp can optionally be compiled with rational numbers support. The following functions are defined even without rational support.

rational -> *<object>* **feature**

rational feature is defined if the current implementation supports rational numbers.

(rational *number***)** -> *<number>* **function**
(rationalize *number***)** -> *<number>* **function**

Each of these functions converts any number to a rational number. If the argument is already rational, it is returned. *Without rational support, these functions are equivalent to* **truncate**.

(numerator *rational***)** -> *<integer>* **function**
(denominator *rational***)** -> *<integer>* **function**

These functions take a rational number and return as an integer the numerator or denominator of the canonical reduced form of the rational. The numerator of an integer is that integer; the denominator of an integer is **1**.

(/ *number+***)** -> *<number>* **function**

When implementation supports rational numbers, **/** will produce a ratio if the mathematical quotient of two integers is not an exact integer.

Examples:

```
(/ 12 4)        -> 3
(/ 13 4)        -> 13/4
(/ -8)          -> -1/8
(/ 3 4 5)       -> 3/20
```

26 Class Functions

As Common Lisp compatible extension, **OpenLisp** **defclass** macro supports **:allocation** slot option. It controls the kind of slot that is defined. If the value of the **:allocation** slot option is **:instance** (which is the default), a local slot is created. If the value of **:allocation** is :**class,** a shared slot is created. A shared slot defined by a class is accessible in all instances of that class **:class** :**allocation** slot declaration option.

A shared slot can be shadowed. For example, if a class C1 defines a slot named S whose value for the **:allocation** slot option is **:class**, that slot is accessible in instances of C1 and all of its subclasses. However, if C2 is a subclass of C1 and also defines a slot named S, C1's slot is not shared by instances of C2 and its subclasses. When a class C1 defines a shared slot, any subclass C2 of C1 will share this single slot unless the **defclass** form for C2 specifies a slot of the same name or there is a superclass of C2 that precedes C1 in the class precedence list of C2 that defines a slot of the same name.

Example:

```
(defclass <person> ()
  ((name :initarg :name :accessor name :allocation :instance) ;; default
   (species
      :initform 'homo-sapiens
      :accessor species
      :allocation :class)))
```

(create-class *class-name super-class***)** -> *<class>* **function**

Returns a new class object with name *class-name* that is a subclass of *super-class*. User classes must inherit, directly or indirectly of **<standard-class>** class. System classes must inherit, directly or indirectly of **<built-in-class>** class.

(class-name *class***)** -> *<class>* **function**

The function **class-name** takes a class object and returns its name. The *class* argument is a class object. The name of the given class is returned.

(class-metaclass *class***)** -> *<class>* **function**

The function **class-metaclass** takes a class object and returns its metaclass. The *class* argument is a class object. The metaclass object of the given class is returned.

(class-precedence-list *class***)** -> *<list>* **function**

The function **class-precedence-list** takes a class object and returns its class precedent list. The *class* argument is a class object.

(class-abstract-p *class***)** -> *<boolean>* **function**

The function **class-abstract-p** takes a class object and returns **t** if the class argument is an abstract-class class object (i.e. we the class is non-instanciable), or **nil** otherwise.

(class-size *class***)** -> *<integer>* **function**

The function **class-size** takes a class object and returns the size of element for this class.

(class-direct-superclasses *class***)** -> *<list>* **function**

The function **class-direct-superclasses** takes a class object and returns the list of superclasses of this class.

(class-initargs *class***)** **->** *<list>* **function**

---

class-initargs is an internal function that returns the names of initargs of the given class. It returns **nil** if no information is available for this class.

(class-slot-descriptions *class***)** -> *<list>* **function**

The function **class-slot-descriptions** takes a class object and returns the list of its slot descriptions.

Example:

```
(defclass <foo> ()
   ((x :initform 0
       :accessor yab-x
       :initarg  x)))                  => <foo>

(defclass <bar> (<foo>)
   ((y :accessor yab-y
       :initarg  y-value)))            => <bar>

(class-direct-superclasses (class <foo>))  => nil
(class-direct-superclasses (class <bar>))  => (<foo>)
(class-precedence-list <foo>)              => (<foo>
```

```
                                             <standard-object>
                                             <object>)
(class-precedence-list <bar>)             => (<bar>
                                             <foo>
                                             <standard-object>
                                              <object>)
(class-size (class <foo>))                => 1 ;; one local slot
(class-size (class <bar>))                => 2 ;; one local slot + one inherited slot.
(class-initargs (class <foo>))            => (x)
(class-initargs (class <bar>))            => (x y-value)
(class-slot-descriptions (class <foo>))   => ((x :initform 0
                                                 :accessor yab-x
                                                 :initarg  x))
(class-slot-descriptions (class <bar>))   => ((y :accessor yab-y
                                                 :initarg  y-value))
```

(class-shared-slots *class***)** -> *<list>* **function**

The function **class-shared-slots** takes a class object and returns the list of its shared slots.

(find-class *symbol* **[***errorp***])** -> *<object>* **function**

The first argument to **find-class** is a symbol. If there is no such class and the *errorp* argument is not supplied or is non-**nil**, **find-class** signals an error. If there is no such class and the *errorp* argument is **nil**, **find-class** returns **nil**. The default value of *errorp* is **t**.
The result of **find-class** is the class object named by the given symbol.
The class associated with a particular symbol can be changed by using **setf** with **find-class**. The results are undefined if the user attempts to change the class associated with a symbol that is defined as a built-in type.

(create-class-info *fileds initforms initargs slot-spec sc-list***)** -> *<vector>* **function**

create-class-info is an internal function that creates a class info structure using supplied values.

(allocate-object *class***) ->** *<object>* **function**

Returns a new instance of class *class* where all slots are non-initialize (*i.e.* #**<unbound-value>**). This instance is generally used by the generic function **initialize-object**.

(default-initialize-object *instance keys class***) ->** *<object>* **function**

This function, called by primary method of generic function **initialize-object**, initialize instance *instance* with *keys* which list **((***keyword$_1$ value$_1$***)** … **((***keyword$_N$ value$_n$***))**. Third parameter *class*, if non-**nil**, must be the class of *instance*.

(slot-value *object slot***)** -> *<object>* **function**
(set-slot-value *value object slot-name***)** -> *<object>* **function**
(setf (slot-value *object slot-name***)** *value***)** -> *<object>* **function**

The function **slot-value** returns the value contained in the slot *slot-name* of the given object. If there is no slot with that name, **slot-missing** is called. If the slot is unbound, **slot-unbound** is called. The macro **setf** can be used with **slot-value** to change the value of a slot (calling **set-lot-value** function). The arguments are the object and the name of the given slot. The result is the value contained in the given slot.

(slot-boundp *object slot***)** -> *<boolean>* **function**

The function **slot-boundp** tests whether a specific slot in an instance is bound. The arguments are the instance and the name of the slot. The function **slot-boundp** returns true or false. This function allows for writing **:after** methods on **initialize-object** in order to initialize only those slots that have not already been bound.

(slot-index *instance index***)** -> *<object>* **function**

This internal function returns the slot value at position *index* for the object *object.* If this slot is unbound, the generic function **slot-unbound** is called with two arguments *instance* and *slot*. The default behavior of **slot-unbound** is to signal an error *slot-unbound*.

(set-slot-index *value instance slot***)** -> *<object>* **function**
(setf (slot-index *instance index***)** *value***)** -> *<object>* **function**

This internal function and the **seft** form return the slot value at position *index* for the object *object.* If this slot is unbound, the generic function **slot-unbound** is called with two arguments *instance* and *slot*. The default behavior of **slot-unbound** is to signal an error *slot-unbound*.

(slot-makunbound *instance slot-name***)** -> *<object>* **function**

The function **slot-makunbound** restores a slot in an instance to the unbound state. The arguments to **slot-makunbound** are the instance and the name of the slot. The instance is returned as the result.

(slot-exists-p *object slot-name***)** -> *<object>* **function**

The function **slot-exists-p** tests whether the specified object has a slot of the given name. The *object* argument is any object. The *slot-name* argument is a symbol. The function **slot-exists-p** returns true or false.

(slot-unbound *class instance slot-name***)** -> *<object>* **generic function**

The generic function **slot-unbound** is called when an unbound slot is read in an instance whose metaclass is **<standard-class>**. The default method signals an error. The generic function **slot-unbound** is not intended to be called by programmers. Programmers may write methods for it. The function **slot-unbound** is called only by the function <**slot-value**. The arguments to **slot-unbound** are the class of the instance whose slot was accessed, the instance itself, and the name of the slot. If a method written for **slot-unbound** returns values, these values get returned as the values of the original function invocation. An unbound slot may occur if no **:initform** form was specified for the slot and the slot value has not been set, or if **slot-makunbound** has been called on the slot.

(print-object *class instance stream***)** -> *<object>* **generic function**

The generic function **print-**object writes the printed representation of *class-instance* to *stream*. The function **print-object** is called by the **OpenLisp** printer; it should not be called by the user.

Each implementation is required to provide a method on the class **<standard-object>** and on the class **<standard-structure>**. There must be always an applicable method. Users may write methods for **print-object** for their own classes if they do not wish to inherit an implementation-dependent method.

The method on the class **<standard-structure>** prints the object in the default **#s(**...**)** notation;

(print-unreadable-object (obj st **:type** x **:identity** y*) . body)* -> *<object>* **macro**

The macro **print-unreadable-object** outputs a printed representation of object *obj* on stream *st*, beginning with '**#<**' and ending with '**>**'. Everything output to stream by the *body* body forms is enclosed in the angle brackets. If optional **:type** argument is **true**, the output from forms is preceded by a brief description of the object's type and a space character. If optional **:identity** argument is **true**, the output from forms is followed by a space character and a representation of the object's identity, typically a storage address.

Example:

```
(defmethod print-object ((obj <myobj>) stream)
   (print-unreadable-object (obj stream :type t :identity t)
   (format stream "myobj: ~s" (id obj))))
```

27 Streams Functions

(format *stream fmt exp$_1$ .. exp$_N$*) -> *<object>* **function**

As an extension to the ISLISP standard, **OpenLisp** provides the following directives to the **format** function.

ISLISP :

~w[,d]G	print the next argument (a float) with *w* characters and *d* characters after the dot.
~w[,d]E	print the next argument (a float) with *w* characters and *d* characters after the dot.
~w[,d]F	print the next argument (a float) with *w* characters and *d* characters after the dot.
~wD	print the next argument (an integer) with *w* characters after the dot.
~[:[@]]P	plural. If arg is not **eql** to the integer 1, a lowercase s is printed; if arg is **eql** to 1, nothing is printed. (Notice that if arg is a floating-point 1.0, the "*s*" is printed.) **~:P** prints a lowercase "*s*" if the last argument was not 1. **~@P** prints "*y*" if the argument is 1, or "*ies*" if it is not. **~:@P** does the same thing, but backs up first.
~[n][:@]]*	the next arg is ignored. **~*n**** ignores the next n arguments. **~:*** "ignores backwards". **~*n*@*** is an "absolute goto" rather than a "relative goto". It goes to the nth arg, where 0 means the first one; n defaults to 0.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (') followed by the desired character may be used as a prefix parameter, to mean the character object that is the character following the single quote. For example, you can use ~5,'0d to print an integer in decimal radix in five columns with leading zeros, or ~5,'*d to get leading asterisks.

In place of a prefix parameter to a directive, you can put the letter V (or v), which takes an argument from arguments for use as a parameter to the directive. Normally this should be an integer or character object, as appropriate. This feature allows variable-width fields and the like. If the argument used by a V parameter is **nil**, the effect is as if the parameter had been omitted.

(format-user-type *stream object level*) -> *<object>* **function**

This function outputs user objects (as defined by **defclass** or **defstruct**) to the stream *stream. level* is an integer indicating the current depth (to be compared against ***print-level***). The printing function should observe the values of such printer-control variables as ***print-escape***. By default, **format-user-type** prints *objet* as **#<new-type @address>**. **format-user-type** always returns *object.*

(prin *object**) -> *<object>* **function**
(princ *object**) -> *<object>* **function**
(print *object**) -> *<object>* **function**

prin outputs the printed representation of *object** to **(output-stream)**. Escape characters are used as appropriate. Roughly speaking, the output from **prin** is suitable for input to the function **read**. **prin** returns the last *object* as its value. **printc** is just like **prin** except that the output has no escape characters. **print** is like **prin** except that a newline is output after the last argument is printed.

Note : this is a major difference with CLtL equivalent name.

(terpri) -> *<null>* **function**

The function **terpri** outputs a newline to **(output-stream)**. It is identical in effect to **(write-char #\newline** *output-stream***)**; however, **terpri** always returns **nil**.

(write-to-string *object* **[***escape***])** -> *<string>* **function**

The object is effectively printed and the characters that would be output are made into a string, which is returned. If optional argument *escape* is non-**nil**, the string contains escape characters.

(read-from-string *string* **[***eof-error-p* **[** *eof-value***]])** -> *<object>* **function**

The characters of *string* are given successively to the Lisp reader, and the Lisp object built by the reader is returned. Macro characters and so on will all take effect. As with other reading functions, the arguments *eof-error-p* and *eof-value* control the action if the end of the (sub)string is reached before the operation is completed; reaching the end of the string is treated as any other end-of-file event.

(read-delimited-list *char* **[***stream***])** -> *<object>* **function**

This reads objects from *stream* until the next character after an object's representation (ignoring whitespace characters and comments) is *char*. (The char should not have whitespace syntax in the current readtable.) A list of the objects read is returned.

(unread-char *character* **[***input-stream***])** -> *<null>* **function**

---

unread-char puts the *character* onto the front of *input-stream*. The *character* must be the same character that was most recently read from the *input-stream*. The *input-stream* "backs up" over this character; when a character is next read from *input-stream*, it will be the specified character followed by the previous contents of *input-stream*. **unread-char** returns **nil**.
One may apply **unread-char** only to the character most recently read from *input-stream*. Moreover, one may not invoke **unread-char** twice consecutively without an intervening **read-char** operation. The result is that one may back up only by one character, and one may not insert any characters into the *input stream* that were not already there.

(stream-element-class *stream***)** -> *<object>* **function**

Returns the element class used to open *stream*. It returns **nil** if *stream* is not an opened stream.

Example:

```
(with-open-input-file (istream file '<wide-character>)
   (stream-element-class istream))          => <wide-character>
```

(interactive-stream-p *stream***)** -> *<object>* **function**

The intent is to distinguish between interactive and batch (background, command-file) operations. Some characteristics that might distinguish a stream as interactive:

- The stream is connected to a person (or the equivalent) in such a way that the program can prompt for information and expect to receive input that might depend on the prompt.
- The program is expected to prompt for input and to support "normal input editing protocol" for that operating environment.
- A call to read-char might hang waiting for the user to type something rather than quickly returning a character or an end-of-file indication.

28 The Readtable

There is a data structure called the readtable that is used to control the reader. It contains information about the syntax of each character. It is set up to give the standard ISLisp meanings to all the characters, but the user can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the dynamic variable **`*readtable*`**.

`*readtable*` -> *<readtable>* **dynamic variable**

The dynamic value of *readtable* is the current readtable. The initial value of this is a readtable set up for standard ISLisp syntax. You can bind this variable to temporarily change the readtable being used.
To program the reader for a different syntax, a set of functions are provided for manipulating readtables. Normally, you should begin with a copy of the standard ISLisp readtable and then customize the individual characters within that copy.

(copy-readtable [[*from-readtable***]** *to-readtable***])** -> *<object>* **function**

A copy is made of from-readtable, which defaults to the current readtable (the value of the dynamic variable **`*readtable*`**). If *from-readtable* is **nil**, then a copy of a standard ISLisp readtable is made.

For example

```
(setf (dynamic *readtable*) (copy-readtable nil))
```

will restore the input syntax to standard Common Lisp syntax, even if the original readtable has been clobbered (assuming it is not so badly clobbered that you cannot type in the above expression!). On the other hand,

```
(setf (dynamic *readtable*) (copy-readtable))
```

will merely replace the current readtable with a copy of itself.
If *to-readtable* is unsupplied or **nil**, a fresh copy is made. Otherwise, *to-readtable* must be a readtable, which is destructively copied into.

(readtablep *object***)** -> *<boolean>* **function**

readtablep is **true** if its argument is a readtable, and otherwise is false.

(set-syntax-from-char *to-char from-char* **[[***to-readtable from-readtable***]])** -> *<boolean>* **function**

This makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. The *to-readtable* defaults to the current readtable (the value of the global variable **`*readtable*`**), and *from-readtable* defaults to **nil**, meaning to use the syntaxes from the standard ISLisp readtable.
set-syntax-from-char function returns **t**.

(set-macro-character *char function* **[[***non-terminating-p***]** *readtable***])** -> *<object>* **function**
(get-macro-character *char* **[***readtable***])** -> *<function>* **function**

`set-macro-character` causes *char* to be a macro character that when seen by **`read`** causes *function* to be called.
`get-macro-character` returns the function associated with *char*.

The *function* is called with two arguments, *stream* and *char*. The *stream* is the input stream, and *char* is the macro character itself. In the simplest case, *function* may return a Lisp object. This object is taken to be that whose printed representation was the macro character and any following characters read by the *function*. As an example, a plausible definition of the standard single quote character is:

If *non-terminating-p* optional argument is not **`nil`** (it defaults to **`nil`**), then it will be a non-terminating macro character: it may be embedded within extended tokens.

In each case, *readtable* defaults to the current readtable. If *readtable* is **`nil`**, standard readtable is used.

```
(defun single-quote-reader (stream char)
   (list 'quote (read stream t nil)))

(set-macro-character #\' #'single-quote-reader)
```

`(make-dispatch-macro-character` *char* **`[[`***non-terminating-p***`]`** *readtable***`])`** -> <*boolean*>**`function`**

This causes the character *char* to be a dispatching macro character in *readtable* (which defaults to the current readtable). **`make-dispatch-macro-character`** returns **`t`**.

If *non-terminating-p* optional argument is not **`nil`** (it defaults to **`nil`**), then it will be a non-terminating macro character: it may be embedded within extended tokens.

`(set-dispatch-macro-character` *disp-char sub-char function* **`[`***readtable***`])`** -> <*boolean*>**`function`**
`(get-dispatch-macro-character` *disp-char sub-char* **`[`***readtable***`])`** -> <*function*> **`function`**

`set-dispatch-macro-character` causes function to be called when the *disp-char* followed by *sub-char* is read. The *readtable* defaults to the current readtable. The arguments and return values for *function* are the same as for normal macro characters except that function gets *sub-char*, not *disp-char*, as its second argument and also receives a third argument that is the non-negative integer whose decimal representation appeared between *disp-char* and *sub-char*, or **`nil`** if no decimal integer appeared there.
The *sub-char* may not be one of the ten decimal digits; they are always reserved for specifying an infix integer argument. Moreover, if *sub-char* is a lowercase character (see **`lower-case-p`**), its uppercase equivalent is used instead. (This is how the rule is enforced that the case of a dispatch sub-character doesn't matter.)
`set-dispatch-macro-character` returns **`t`**.
`get-dispatch-macro-character` returns the macro-character function for *sub-char* under *disp-char*, or **`nil`** if there is no function associated with *sub-char*. If *readtable* is **`nil`**, standard readtable is used
If the *sub-char* is one of the ten decimal digits 0 1 2 3 4 5 6 7 8 9, **`get-dispatch-macro-character`** always returns **`nil`**. If *sub-char* is a lowercase character, its uppercase equivalent is used instead.

29 Input/Output Files

`(delete-file` *file***`)`** -> <*boolean*> **`function`**

The specified *file* is deleted. The *file* may be a string, a pathname, or a stream. If it is an open stream associated with a file, then the stream itself and the file associated with it are affected (if the file system permits).

`(copy-file` *$file_1$ $file_2$***`)`** -> <*object*> **`function`**

The specified *file1* is copied to *file2* (which must be a file name).

(append-file $file_1$ $file_2$**)** -> *<object>* **function**

The specified *file2* is appended to *file1* (which must be a file name).

(rename-file *file new-name***)** -> *<object>* **function**

The specified *file* is renamed to *new-name* (which must be a file name).

(encode-file *file newfile***)** -> *<object>* **function**

The specified *file* is encode into *newfile* so that it can be loaded using **load-binary**.

(load *filename* **[[***verbose***]** *filemode***])** -> *<object>* **function**

This function loads the file named by *filename* into the Lisp environment. It is assumed that a text (character file) can be automatically distinguished from an object (binary) file by some appropriate implementation-dependent means, possibly by the file type. If *filemode* is **<wide-character>**, the file is opened in binary mode and it assumes that the file contains UNICODE characters. If *filemode* is **<utf8-character>**, it assumes that the file contains UNICODE characters UTF-8 encoded.

(libload *filename* **[***verbose***])** -> *<object>* **function**

This function loads the file named by *filename* into the Lisp environment. It is assumed that a text (character file) can be automatically distinguished from an object (binary) file by some appropriate implementation-dependent means, possibly by the file type. The defaults for *filename* are taken from the dynamic variable ***system-path***. The *verbose* argument (which defaults to the value of dynamic variable ***load-verbose***), if true, permits **libload** to print a message in the form of a comment (that is, with a leading semicolon) to **(standard-output)** indicating what file is being loaded and other useful information.

(load-dynamic-module *filename***)** -> *<object>* **function**

This function loads the compiled module named by *filename* (a DLL in Windows) into the Lisp environment. It returns the handle associated to this module.

(unload-dynamic-module *module-handle***)** -> *<object>* **function**

This function unloads the compiled module associated with *module-handle* from the Lisp environment. After completion, functions previously defined by this module are removed. The behavior is undefined if you try to call a function from a removed module.

(load-binary *filename***)** -> *<object>* **function**

This function loads the file named by *filename* into the Lisp environment. It is assumed that this is a binary file by some appropriate implementation-dependent means, possibly by the file type.

(load-stdlib *cpflag* **[***verbose***])** -> *<object>* **function**

This function loads the standard environment (files may depend on implementation). If *cpflag* is **t**, files are loaded in compiled format. When *verbose* is non-**nil**, the name of loaded files is printed.

(search-in-path *filename* [*path-list*] *ext*]**)** -> *<string>* **function**

Search for filename (a string or a symbol) in the default ***system-path*** dynamic variable directory list or, if supplied, in *path-list* list. If filename was given without extension, .lsp, .lap and .fsl extensions will be used to try to find a valid filename. When a third argument is provided, it names the requested extension. If a valid filename is found; it returns the relative pathname of the file or **nil** if none is found.

```
(search-in-path "foo")  -> nil

;; On a unix system:
(search-in-path "foo" '("../user" "../module"))-> "../module/foo.lap"
(search-in-path "foo" '("../user" "../module") ".c") -> "../module/foo.c"
;; On a windows system:
(search-in-path "foo" '("../user" "../module"))-> "..\module\foo.lap"
(search-in-path "foo" '("../user" "../module") ".c") -> "..\module\foo.c"
```

(local-pathname *path***)** -> *<object>* **function**

Converts portable pathname *path* to local pathname. Portable pathnames are described using ‘unix’ syntax (*i.e.* path are delimited using ‘/’).

```
;; On a unix system:
(local-pathname "/usr/local/openlisp")    -> "/usr/local/openlisp"

;; On a windows system:
(local-pathname "/usr/local/openlisp")    -> "\usr\local\openlisp"
```

(expand-pathanme *path***)** -> *<object>* **function**

On most systems (posix, unix, Windows…), this function returns the list of files in *path* directory. **nil** is returned if this feature is not available or if *path* is not a valid directory.

(current-directory) -> *<stringt>* **function**

On most systems (posix, unix and Windows…), this function returns the current directory.

(directoryp *path***)** -> *<boolean>* **function**

On most systems (posix, unix, Windows…), this function returns **t** only if *path* is a directory.

(change-directory *path***)** -> *<boolean>* **function**

On most systems (posix, unix, Windows…), this function change the current directory.

(create-directory *path***)** -> *<boolean>* **function**

On most systems (posix, unix, Windows…), this function creates a new directory named path. It returns **t** on success and **nil** otherwise.

(remove-directory *path***)** -> *<boolean>* **function**

On most systems (posix, unix, Windows…), this function removes the directory named *path*. The behavior is undefined if the directory is not empty. It returns **t** on success and **nil** otherwise.

(save-core *filename***)** -> *<object>* **function**

On most systems (posix, unix, Windows…), this function saves in the file named *filename* the data of current running lisp image. This image can be restored at later time by **restore-core** function.

(restore-core *filename***)** -> *<object>* **function**

On most systems (posix, unix, Windows…), this function restores from the file named *filename* the data of an old lisp image saved by **`save-core`** function.

`(core-init-std)` -> <*object*> **`function`**

This function is called right after the **`restore-core`** function. It may be redefined to automatically call your own code.

30 System and process functions

`(machine-info)` -> <*general-vector*> **`function`**

On most systems, this function returns a vector describing the hardware platform on which the **OpenLisp** is running on (system name, machine name, version, subversion, processor, openlisp version…). If information is not available for a given slot, its value is **`nil`**.

`(time` *form***`)`** -> <*object*> **`function`**

This evaluates *form* and returns the time spent to evaluate *form.* The nature and format of the information is implementation-dependent. However, implementations are encouraged to provide such information as elapsed real time.

`(sleep` *second***`)`** -> <*object*> **`function`**

`(sleep` *n***`)`** causes execution to cease and become dormant for approximately *n* milliseconds of real time, whereupon execution is resumed. The argument may be any non-negative number. **`sleep`** returns **`nil`**. NOTE: on some systems, the timer may be rounded to the nearest second.

`(alarm` *ms loop***`)`** -> <*boolean*> **`function`**

`alarm` sets a timer that will excute a user redefinable **`clock`** function after *ms* milliseconds. If *loop* parameter is non-**`nil`**, the execution is indefinitely repeated each *ms* milliseconds. NOTE: on some systems, the timer may be rounded to the nearest second.

`(clock)` -> <*object*> **`function`**

`clock` is the default function called when alarm is set (by calling **`alarm`** function). Its aim is to be redefined by user before **`alarm`** is called.

<u>Example:</u>

```
(defun clock ()
   (throw 'timeout))

(defun game ()
   (alarm 60000 nil)
   (catch 'timeout
          (loop-game))
   (print "Game over!"))
```

`(file-mode` *filename***`)`** -> <*integer*> **`function`**

`file-mode` returns the file access mode for file *filename*. The mode should be understood as an octal value *(consult your operating system manual for allowed values)*.

Example:

```
(format (standard-output) "~O" (file-mode "logfile.txt")) -> #o664
```

(set-file-mode *mode filename***)** -> *<object>* **function**
(setf (file-mode *filename***)** *mode***)** -> *<object>* **function**

Change file access mode for file *filename*. The mode should be given as an octal value *(consult your operating system manual for allowed values)*. It returns t on success and **nil** otherwise.

Example:

```
(set-file-mode "logfile.txt" #o664) => t
(setf (file-mode "logfile.txt") #o664)    => t
```

(file-date *filename date tz-flag***)** -> *<date>* **function**

---

file-date returns the file lats modification date for file *filename*. If *date* is already a **<date>** object (as returned by **make-date**), the structure is filled with the values of file last modification date, when set to **nil**, a new **<date>** structure is allocated. If *tz-flag* is **:localtime**, the returned values are for the computer current local time. When *tz-flag* is **:gmt**, the returned values are for GMT.

Example:

```
(format (standard-output) "~A" (file-date "logfile.txt" nil :gmt))
      => "Sun, 14 Aug 2005 12:10:49 GMT"
```

(quit [*n***])** -> *no value is returned* **function**
(end [*n***])** -> *no value is returned* **function**

Returns to the operating system. If an optional integer argument is provided, it is returned to the operating system.

(system-errno) -> *<integer>* **function**

Returns to system errno value of the last system call that returns an error. This value depends of the operating system and the system called.

(getenv *env-var***)** -> *<object>* **function**

The **getenv** function searches the environment list for a string of the form "name=value", and returns the string containing the value for the specified name. If the specified name cannot be found, **nil** is returned.

(putenv *string***)** -> *<boolean>* **function**

The **putenv** function uses the string argument to set environment variable values. The string argument should be a string of the form "name=value". The **putenv** function makes the value of the environment variable name equal to value by altering an existing variable or creating a new one. In either case, the string becomes part of the environment. It returns **t** if the variable has been set **nil** otherwise.

(set-locale *locale***)** -> *<string>* **function**

If a valid *locale* string is given, the locale is changed and it returns the complete locate string name. Using **nil**, it returns the current locate settings and the empty string **""** resets locale to the system default settings. If the locale is invalid, the function returns **nil** current locale settings of the program are not changed.

Example:

```
(set-locale "Fra")      => "French_France.1252"
(set-locale "")         => "French_France.1252"
(set-locale "Japanese") => "Japanese_Japan.932"
(set-locale nil)        =>
"LC_COLLATE=French_France.1252;LC_CTYPE=French_France.1252;LC_MONETARY=Fren
ch_France.1252;LC_NUMERIC=C;LC_TIME=French_France.1252"
```

At startup, locale can be sets automatically by defining an environment variable named **OLLOCALE**. The same result of the above call can be obtained by setting:

```
set OLLOCALE=Fra
set OLLOCALE=Japanese
```

(get-input-code-page) -> *<integer>* **function**

This function returns the current console input code page or **nil** if current input locale is unknown.

(set-input-code-page *code-page***)** -> *<boolean>* **function**

This function tries to change the console current input code page to the integer value *code-page*. It returns **t**. on success or **nil** if current input locale is not changed.

(get-output-code-page) -> *<integer>* **function**

This function returns the current console output code page or **nil** if current input locale is unknown.

(set-output-code-page *code-page***)** -> *<boolean>* **function**

This function tries to change the current console output code page to the integer value *code-page*. It returns **t**. on success or **nil** if current input locale is not changed.

(line-argument [*n*]**)** -> *<object>* **function**
(system-argument [*n*]**)** -> *<object>* **function**

line-argument returns, when available, the nth string argument passed from the command line interpreter. In general, argument 0 is the command itself. If *n* exceeds the bounds of valid argument **nil** is returned. With no arguments, **line-argument** returns a vector of all arguments. That way, **(length (line-argument))** is an idiom that computes the total number of arguments. When a Lisp file is given, it is interpreted as the command parameter.

system-argument is much like **line-argument** but never interprets program when a lisp file is passed as argument. It returns the exact vector of arguments as passed on command line.

If you launch **OpenLisp** with:

```
% openlisp --cons 100 foo.lsp a b
```

you'll get:

```
(line-argument 0)            => foo.lsp
(line-argument 1)            => "a"
(line-argument 2)            => "b"
(line-argument 3)            => nil
(line-argument)              => #("foo.lsp" "a" "b")
(length (line-argument))     => 3

(system-argument 0)          => openlisp
```

```
(system-argument 1)            => -cons
(system-argument 2)            => 100
(system-argument 3)            => foo.lsp
(system-argument 4)            => "a"
(system-argument 5)            => "b"
(system-argument 6)            => nil
(system-argument)              => #(openlisp -cons 100 foo.lsp "a" "b")
(length (system-argument))     => 6
```

In UNIX style, the **$** macro-character uses **line-argument** returned values and is defined as follow:

- ***** Expands to the positional parameters, starting from one as a vector. That is, $* is equivalent to #($1 $2...).
- **@** Expands to the positional parameters, starting from one as a list. That is, $@ is equivalent to ($1 $2...).
- **#** Expands to the number of positional parameters in decimal.
- **$** Expands to the process ID of the shell (or 1000 if not supported by the OS).
- **0** Expands to the name of the shell script. This is set at shell initialization.
- *var* Expands to the nth parameter of the value of *var*.

With the same example as above, you'll get:

```
$*    => #("a" "b")
$@    => ("a" "b")
$#    => 2
$$    => 12642
$0    => "foo.lsp"
$1    => "a"
$2    => "b"
```

(system-argument [*n*]**)** -> <*string*> **function**

system-argument is much like **line-argument** but never interprets program when a lisp file is passed as argument. It returns the exact vector of arguments as passed on command line.

(system-name) -> <*symbol*> **function]**

Returns the symbol corresponding of the operating system (examples : **unix**, **posix**, **ms-dos**, **windows**, **nt**, **os2**, ...). This value can be used with #- or #+ reader directives.

(comline *cmd***)** -> <*integer*> **function**

This function passes the string *cmd* to the underlining operating system and returns the value that is returned by the command interpreter. It returns the value 0 only if the command interpreter returns the value 0. A return value of – 1 indicates an error.

(spawn *fmt* **[***arg_1 ... arg_n***])** -> <*integer*> **function**

This function combines the format string *fmt* and arguments *arg_i*. Then, **spawn** passes the resulting string to the underlining operating system and returns the value that is returned by the command interpreter. It returns the value 0 only if the command interpreter returns the value 0. A return value of – 1 indicates an error.

(spawn "cp ~A ~A.BAK" file file) => 0

(fork) -> <*integer*> **function**

UNIX Only!! The **fork** function creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited. On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created.

`(wait)` -> *<integer>* **`function`**

UNIX Only!! The **`wait`** function suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed. It returns the process ID of the child which exited or -1 on error. It also returns –1 if this feature is not available (*i.e.* WIN32).

`(waitpid` *<pid>* **`[`***<nohang>***`])`** -> *<integer>* **`function`**

The **`waitpid`** function suspends execution of the current process until a child as specified by the *<pid>* argument has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed. It returns the process ID of the child which exited or -1 on error. It also returns –1 if this feature is not available.

The value of *<pid>* can be one of:

< -1	to wait for any child process whose process group ID is equal to the absolute value of pid.
-1	to wait for any child process; this is the same behavior which wait exhibits.
0	to wait for any child process whose process group ID is equal to that of the calling process.
> 0	to wait for the child whose process ID is equal to the value of *<pid>*.

If the optional argument *nohang* is non-**`nil`** this functions returns immediately if no child has exited.

`(getpid)` -> *<integer>* **`function`**

`getpid` returns the process ID of the current process. (This is often used by routines that generate unique temporary file names.). It returns 1000 if this feature is not available.

`(getid)` -> *<integer>* **`function`**

UNIX Only!! **`getuid`** returns the real user ID of the current process. It returns **`-1`** if this feature is not available (*i.e.* WIN32).

`(setuid` *<uid>***`)`** -> *<boolean>* **`function`**

UNIX Only!! **`setuid`** sets the effective user ID of the current process. If the effective userid of the caller is root, the real and saved user ID's are also set. It returns **`t`** on success and **`nil`** on error. It also returns **`nil`** if this feature is not available (*i.e.* WIN32).

`(kill` *<pid> <sig>***`)`** -> *<integer>* **`function`**

UNIX Only!! The **`kill`** function can be used to send any signal to any process group or process. It returns 0 on success and –1 on error. It also returns –1 if this feature is not available.

if *<pid>* is positive, then signal *<sig>* is sent to *<pid>*.
if *<pid>* equals 0, then *<sig>* is sent to every process in the process group of the current process.
if *<pid>* equals -1, then *<sig>* is sent to every process except for the first one.
if *<pid>* is less than -1, then *<sig>* is sent to every process in the process group **`(abs`** *<pid>***`)`**.
if *<sig>* is 0, then no signal is sent, but error checking is still performed.

Windows Only!! The **`kill`** function can be used to terminate process having a PID equals to *<pid>*. If allowed, the process will exit with *<sig>* as return code.

(run-as-daemon [*outfile*}) -> *<boolean>* **function**

UNIX Only!! The **run-as-daemon** function can be used to run the **OpenLisp** process as a daemon process. Without parameter, standard streams are closed and reopened to a null stream (/dev/null). If an optional string *outfile* is provided, **(standard-output)** and **(error-output)** streams will be redirected to that file. It returns **t** on success and **nil** on error. It also returns **nil** if this feature is not available (*i.e.* WIN32). Alternatively, this feature is provided by launching **OpenLisp** with the **–D** flag.

(daemonp) -> *<boolean>* **function**

UNIX Only!! The **daemonp** function returns **t** if the current image runs as a daemon process or **nil** otherwise.

(get-priority *<which> <who>***)** -> *<integer>* **function**
(set-priority *<which> <who> <prio>***)** -> *<integer>* **function**

UNIX Only!! The scheduling priority of the process, process group, or user, as indicated by *<which>* and *<who>* is obtained with the **get-priority** call and set with the **set-priority** call. *<which>* is one of 0 (PRIO_PROCESS), 1 (PRIO_PGRP), or 2 (PRIO_USER), and *<who>* is interpreted relative to *<which>* (a process identifier for PRIO_PROCESS, process group identifier for PRIO_PGRP, and a user ID for PRIO_USER). A zero value for *<who>* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. For **set-priority** call, *<prio>* is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

Windows Only!! The scheduling priority of the process is obtained with the **get-priority** call and set with the **set-priority**. *<which>* and *<who>* are ignored on Windows. For **set-priority** call, *<prio>* is a value in the range -20 to 20. The default priority is 0; lower priorities cause more favorable scheduling.

31 Socket Streams Functions

A socket is a communication endpoint — an object through which a Sockets application sends or receives packets of data across a network. A socket has a type and is associated with a running process, and it may have a name. Currently, sockets generally exchange data only with other sockets in the same "communication domain," which uses the Internet Protocol Suite. Both kinds of sockets are bi-directional: they are data flows that can be communicated in both directions simultaneously (full-duplex). Two socket types are available:
Stream sockets provide for a data flow without record boundaries — a stream of bytes. Streams are guaranteed to be delivered and to be correctly sequenced and unduplicated.
Datagram sockets support a record-oriented data flow that is not guaranteed to be delivered and may not be sequenced as sent or unduplicated.
"Sequenced" means that packets are delivered in the order sent. "Unduplicated" means that you get a particular packet only once.

ISLISP streams (**<stream>** class) have been extender to support sockets (**<socket>** class) as a subtype of streams. This way, we can use standard I/O functions that need **<stream>** with **<socket>** (**read**, **read-byte**, **read-char**, …) for input and (**write-char**, **print**, **format-xxx**) for output.

The current C implementation uses POSIX when available *« Protocol Independent Interface (PII) - P1003.1g »* or WinSock on Windows systems.

socket -> *<object>* **feature**

socket feature is defined if the current implementation supports sockets.

default-ip-version -> *<symbol>* **dynamic variable**

This dynamic variable, which is either **:ipv4** (default) or **:ipv6**, contains the default value used for IP addresses.

(socketp *object***)** -> *<boolean>* **function**

Returns **t** si *object* is a socket (**<socket>** class) or **nil** otherwise.

(socket [*type* **[***ipver***])** -> *<socket>* **function**

socket function creates a socket that is bound to a specific service provider. This function returns **nil** if the socket has not been created. The *type* parameter is either **:tcp** or **:udp**. The *ipver* parameter is either **:ipv4** (default) or **:ipv6**.

(socket-address *type port address***)** -> *<socket-address>* **function**

socket-address function creates a internet address where *type* is either **:ipv4** or **:ipv6**, *port* is the port and address is the internet address. The return value may be use to **send-to**.

(bind *socket ip port***)** -> *<socket>* **function**

The **bind** function shall assign a local socket address *ip* and port *port* to a socket identified by descriptor *socket* that has no local socket address assigned. Sockets created with the **socket** function are initially unnamed; they are identified only by their address family. When *ip* is **t**, it binds the socket to all available interfaces *(i.e.* **INADDR_ANY***)*. If operation succeeds, it returns *socket* and raises an error otherwise.

(connect *socket netaddr service protocol* **[***ipver***])** -> *<boolean>* **function**

connect function establishes a connection (client interface) to a specifed unconnected socket *socket* with the service *service* at address *netaddr* and using the protocol *protocol.* If *service* is passed as an integer it is used as the port number that will be used by the socket. In that case, *protocol* is simply ignored. If *protocol* is **nil**, the first service found with *service* name is used. The *ipver* parameter is either **:ipv4** (default) or **:ipv6**. The function returns **t** if the socket *socket* is correctly connected or **nil** otherwise. Service names are set in a file names service which can be in directory **/etc** or **/etc/inet** for UNIX Systems. On Windows, you can find it in **\windows\system32\drivers\etc**. For example, if there is a service named « testtcp » using protocol « tcp », you can connect to a server at adress IP 193.57.0.1 with :

Example:

```
;; services contains:
testtcp     8192/tcp

;; in Lisp:

(let ((client (socket)))
     (connect client "193.57.0.1" "testtcp" "tcp")
     ;; talk to the server
     )

;; we can also call port 8192

(let ((client (socket)))
     (connect client "193.57.0.1" 8192)
     ;; talk to the server
     )
```

(listen *socket* **[***backlog***])** -> *<integer>* **function**

To accept connections, a socket is first created with the **socket** function and willing to accept incoming connexions with **listen**. Then the connections are accepted with the **accept** function. The *backlog* parameter defines the maximum length for the queue of pending connections. If not supplied, the system maximal value is used. Sockets that are connection oriented are used with **listen**. The socket *socket* is put into "passive" mode where incoming connection requests are acknowledged and queued pending acceptance by the process.
The **listen** function is typically used by servers that can have more than one connection request at a time. If a connection request arrives and the queue is full, the client will receive an error. If there are no available socket descriptors, **listen** attempts to continue to function. If descriptors become available, a later call to **listen** or **accept** will refill the queue to the current or most recent "backlog", if possible, and resume listening for incoming connections.
An application can call **listen** more than once on the same socket. This has the effect of updating the current backlog for the listening socket. Should there be more pending connections than the new *backlog* value, the excess pending connections will be reset and dropped.

Example :

```
;; services contains :
testtcp     8192/tcp

;; in Lisp :

(let ((server (socket :tcp)))
     (listen server)
     ;; server can accept clients
     )
```

(select *n infd readfd writefd exceptfd timeo***)** -> *<integer>* **function**

The **select** function is used to determine the status of one or more sockets. For each socket, the caller can request the *n* first informations on read, write or error status. The set of sockets for which a given status is requested is indicated by *infd* vector. The sockets contained within the *infd* structures must be associated with a single service provider. Upon return, the vectors are updated to reflect the subset of these sockets that meet the specified condition. The **select** function returns the number of sockets meeting the conditions. A set of vectors is provided for manipulating an *infd* vector. The parameter *readfds* identifies the sockets that are to be checked for readability. If the socket is currently in the **listen** state, it will be marked as readable if an incoming connection request has been received such that an **accept** is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading such that a call to **receive** or **receive-from** is guaranteed not to block.
For connection-oriented sockets, readability can also indicate that a request to close the socket has been received from the peer. If the virtual circuit was closed gracefully, then a **receive** will return immediately with zero bytes read. If the virtual circuit was reset, then a **receive** will complete immediately with an error code. The parameter *writefds* identifies the sockets that are to be checked for writability. If a socket is processing a **connect** call (nonblocking), a socket is writable if the connection establishment successfully completes. If the socket is not processing a **connect** call, writability means a **send** or **send-to** are guaranteed to succeed.
The parameter *exceptfds* identifies the sockets that are to be checked for the presence of out-of-band data or any exceptional error conditions.
Any of the parameters, *readfds*, *writefds*, or *exceptfds*, can be given as nil. At least one must be non-nil, and any non-nil descriptor set must contain at least one handle to a socket. *timeout* is the maximum time for **select** to wait, or **nil** for blocking operation.

```
(let ((select-vector (create-vector 8 ()))
       (read-vector   (create-vector 8 ()))
       (write-vector  (create-vector 8 ())))
      (select-clear select-vector)
      (select-add sock1 select-vector)
```

```
(select 1
        select-vector
        read-vector
        write-vector
        ()
        5.0)
(select-remove sock1 select-vector))
```

(select-clear *socket-vector***)** -> *<object>* **function**

This function clears the vector *socket-vector* with **nil** in all entries.

(select-add *socket socket-vector***)** -> *<boolean>* **function**

This function adds the socket *socket* to vector *socket-vector* and returns **t** if *socket* has been set in *socket-vector* or **nil** if *socket-vector* is full or if *vector* was already in *socket-vector*.

(select-remove *socket socket-vector***)** -> *<boolean>* **function**

This function removes the socket *socket* from vector *socket-vector* and returns **t** if *socket* has been removed from *socket-vector* or **nil** if *vector* was not in *socket-vector*.

(select-test *socket socket-vector***)** -> *<boolean>* **function**

This function tests if the socket *socket* is set in vector *socket-vector*. It returns **t** if *socket* is in *socket-vector* or **nil** otherwise.

(poll *nfds fds timeout[readfds [writefds]]***)** -> *<integer>* **function**

poll performs a similar task to **select**: it waits for one of a set of descriptors to become ready to perform I/O. The set of descriptors to be monitored is specified in the *fds* lisp vector. The timeout argument specifies the number of milliseconds that **poll** should block waiting for a file descriptor to become ready. The call will block until either: a socket descriptor becomes ready; the call is interrupted by a signal handler; or the timeout expires. When supplied and not-**nil**, *readfds* and *writefds* are lisp vector having the same size as *fds* which are filed on return by descriptors having pending read (in *readfds*) or pending write (in *writefds*). **poll** returns the total number of pending I/O in both *readfds* and *writefds*. When *timeout* expires, **poll** return 0.

```
(defglobal *service*  8192)
(defglobal *protocol* nil)
(defglobal *host*     (get-host-address (get-host-name)))
(defglobal *max-fd*   8)

(defun simple-server-with-poll ()
   (with-server-socket (server *service* *protocol*)
     (let ((fdinput (create-vector *max-fd* ()))
           (fdread  (create-vector *max-fd* ()))
           (fdwrite (create-vector *max-fd* ())) ;; not used here
           (res     ()))
          (with-client-socket (client *host* *service* *protocol*)
                ;; send info to server
                (send client "Foo " 4)
                (send client "Bar " 4)
                (format client "Gee ")
                (setf (elt fdinput 3) server)
                ;; launch server and get info.
                (case (poll *max-fd* fdinput 100 fdread)
                      ((-1) 'error)
```

```
                      ((0)  'time-out)
                      (t    (for ((i 0 (1+ i)))
                                 ((>= i *max-fd*))
                                 (when (elt fdread i)
                                       (let ((s (accept server)))
                                            (setq res (list (read s)
                                                            (read s)
                                                            (read s)))
                                            (close s))
                                       (shutdown server 2))
                                 (when (elt fdwrite i)
                                       (format t "Write pending on ~a~%"
                                               server)))
                            res))))))

(simple-server-with-poll)     => (foo bar gee)
```

(accept *socket***)** -> *<socket>* **function**

The **accept** function (server interface) extracts the first connection on the queue of pending connections on socket *socket.* It then creates a new socket and returns a handle to the new socket. The newly created socket is the socket that will handle the actual the connection and has the same properties as socket *socket.*

Example:

```
(let ((server (socket))
      (client ()))
     (listen server "testtcp" "tcp" "193.57.0.1")
     ;; server can accept new clients
     (setq client (accept server))
     ;; connection between server and client
     )
```

(send *socket buffer* **[***len***])** -> *<integer>* **function**
(send-to *socket buffer* **[***len* **[***to***]])** -> *<integer>* **function**

The **send** (:tcp) and **send-to** (:tcp or :udp) functions are used to write outgoing data in *buffer* with *len* bytes on a connected socket *socket*. If *len* is not supplied, the buffer length is used. For message-oriented sockets, care must be taken not to exceed the maximum packet size of the underlying provider, which can be obtained by using getsockopt. If no buffer space is available within the transport system to hold the data to be transmitted, **send** will block unless the socket has been placed in a nonblocking mode. On nonblocking stream-oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both client and server machines. Both functions return the length of data send from *buffer*. If *to* is given and non-**nil**, it must be a valid **<socket-address>** object where *buffer* is sent to.

(receive *socket buffer* **[***len***])** -> *<integer>* **function**
(receive-from *socket buffer* **[***len***])** -> *<integer>* **function**

The **receive** "tcp" and **receive-from** "udp" functions are used to read incoming data on connection-oriented sockets, or connectionless sockets. When using a connection-oriented protocol, the sockets must be connected before calling **receive**. When using a connectionless protocol, the sockets must be bound before calling **receive**.
For connection-oriented sockets, calling **receive** will return as much information as is currently available – up to the size of the buffer or *len* if this optional argument is supplied. Both functions return the length of data read in *buffer*.

(subscribe-multicast-group *socket ip port***)** -> *<integer>* **function**

Joins the socket to the supplied multicast group on *ip port*. The *ip* address must belong to a valid multicast address *(i.e. in the range 224.0.0.0 through 239.255.255.255)*

(close *socket***)** -> <*boolean*> **function**

Since sockets inherit from <stream>, the ISLISP **close** function has been extended to also close the socket *socket*.

(shutdown *socket how***)** -> <*object*> **function**

The **shutdown** function is used on all types of sockets to disable reception, transmission, or both for the socket *socket*. If the *how* parameter is 0 (**SD_RECEIVE** in C), subsequent calls to the **receive** function on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP sockets, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated. If the *how* parameter is 1 (**SD_SEND** in C), subsequent calls to the **send** function are disallowed. For TCP sockets, a FIN will be sent. Setting *how* to 2 (**SD_BOTH** in C) disables both sends and receives as described above. The **shutdown** function does not close the socket. Any resources attached to the socket will not be freed until **close** is invoked. To assure that all data is sent and received on a connected socket before it is closed, an application should use **shutdown** to close connection before calling **close**.

(get-host-name) -> <*string*> **function**

The **get-host-name** function returns the name of the local host. The host name is returned as a string. The form of the host name is dependent on the Sockets provider — it can be a simple host name, or it can be a fully qualified domain name.

(get-host-address *host* **[***ipver***])** -> <*string*> **function**

Return a new string which is the first IP address found for the host. The *ipver* parameter is either **:ipv4** (default) or **:ipv6**.

Example:

```
(get-host-address "coltrane" :ipv4)
      => "193.57.0.1"
(get-host-address "coltrane" :ipv6)
      => "fe80::2cae:14c:19e1:c8ec"
```

(get-host-address-list *host* **[***ipver***])** -> <*list*> **function**

Return a list of all IP address strings for the host. The *ipver* parameter is either **:ipv4** (default) or **:ipv6**.

Example:

```
(get-host-address-list "coltrane" :ipv4)
      => ("193.57.0.1" "192.168.111.1" "192.168.42.1")
(get-host-address-list "coltrane" :ipv6)
      => ("fe80::2cae:14c:19e1:c8ec"
          "fe80::c572:15f7:8d6c:787b
          "fe80::b021:4c3a:bb6a:ec51"
          "2002:c139:1::c139:1")
```

(get-proto-by-name *protocol***)** -> <*string*> **function**

Return a new string that is the port number for protocol *protocol*.

(get-sock-name *socket***)** -> *<string>* **function**

Return a new string that is the IP address for the computer using *socket*.

(get-peer-name *socket***)** -> *<string>* **function**

Return a new string that is the IP addresse for the computer using *socket*.

(net-to-host-short *num***)** -> *<integer>* **function**
(host-to-net-short *num***)** -> *<integer>* **function**
(net-to-host-long *num***)** -> *<integer>* **function**
(host-to-net-long *num***)** -> *<integer>* **function**

Those 4 functions are used to convert 16 bits (short) and 32 bits (long) *num* integers between local machine (host) and remote (net).

(socket-nonblocking *socket flag***)** -> *<boolean>* **function**

When available, **socket-nonblocking** sets the socket *socket* in non-blocking mode if *flag* is **t** or in blocking mode when *flag* is **nil**. It returns **t**, only if operation succeeds. By default, sockets are created in blocking mode and **accept** creates a new socket with the same properties as the connected socket.

```
;; set server to nonblocking
(socket-nonblocking server t)
(setq client (accept server))
;; reset server and client to blocking
(socket-nonblocking server nil)
(socket-nonblocking client nil)
```

(socket-ip-version *socket***)** -> *<symbol>* **function**

Retuns the ip version (either **:ipv4** or **:ipv6**) of socket *socket*.

(socket-keepalive *socket flag***)** -> *<boolean>* **function**

When available, **socket-keepalive** sets the socket *socket* to periodically pool the remote host if *flag* is **t** or remove this mode when *flag* is **nil**. This option causes a packet (called a 'keepalive probe') to be sent to the remote system if a long time (by default, more than 2 hours) passes with no other data being sent or received. This packet is designed to provoke an ACK response from the peer. This enables detection of a peer which has become unreachable (e.g. powered off or disconnected from the net). It returns **t**, only if operation succeeds. By default, sockets are not created in this mode.

```
(setq client (accept server))
;; set client to pool server
(socket-keepalive client t)
...
(socket-keepalive client nil)
```

(socket-multicast-loopback *socket flag***)** -> *<boolean>* **function**

When available, **socket-multicast-loopback** controls whether data sent by an application on the local computer (not necessarily by the same socket) in a multicast session will be received by a socket joined to the multicast destination group on the loopback interface. A value of t causes multicast data sent by an application on the local computer to be delivered to a listening socket on the loopback interface. A value of nil prevents multicast data sent by an application on the local computer from being delivered to a listening socket on the loopback interface. It is enabled by default. It returns **`t`**, only if operation succeeds.

`(socket-nodelay` *socket flag*) -> *<boolean>* **`function`**

When available, **`socket-nodelay`** specifies whether TCP socket *socket* should follow the Nagle algorithm for deciding when to send data. By default, sockets are not created in this mode. The Nagle algorithm says that we should delay sending partial packets in hopes of getting more data. There are bad interactions between persistent connections and Nagle`s algorithm that have very severe performance penalties. Setting flag to **`t`** force TCP to always send data immediately. It should be used when there is an application using TCP for a request/response. To return to default behavior set *flag* to **`nil`**. This function returns **`t`**, only if operation succeeds.

```
(setq client (accept server))
;; set client to pool server
(socket-nodelay client t)
...
(socket-nodelay client nil)
```

`(socket-receive-timeout` *socket timeout*) -> *<boolean>* **`function`**

When available, **`socket-receive-timeout`** sets the socket *socket* receive timeout to *timeout* seconds. It returns **`t`**, only if operation succeeds.

```
(socket-receive-timeout client 1.0)
```

`(socket-send-timeout` *socket timeout*) -> *<boolean>* **`function`**

When available, **`socket-send-timeout`** sets the socket *socket* receive timeout to *timeout* seconds. It returns **`t`**, only if operation succeeds.

```
(socket-send-timeout client 1.0)
```

`(with-client-socket (`<u>*socket*</u> *netaddr service protocol* **`[`***ipver***`])`** *forms**) -> *<object>* **`macro`**

This macro creates a new *socket* and connect it to the service *service* using protocole *protocol* at address *netaddr*. The *ipver* parameter is either **`:ipv4`** (default) or **`:ipv6`**. The the forms *forms* are executed in order. The last evaluated form is returned and the socket is automatically closed. This macro is very used to talk easily with a server using the standard input/ouput functions.

<u>Example:</u>

```
(defmacro with-client-socket (socket &rest body)
  `(let ((,(car socket) (socket)))
        (when ,(car socket)
              (unwind-protect (when ,`(connect ,@socket)
                                    ,@body)
                              (close ,(car socket))))))

(defun simplest-client (exp)
   ;; Client side
   (with-client-socket (st "193.57.0.1" "testtcp" "tcp")
        ;; send an expression to the server
        (format st "~a~%" exp)
        ;; read the result
        (let ((res ))
              (while (not (eq res 'eof))
```

```
                    (setq res (read st () 'eof))
                    (unless (eq res 'eof)
                            (print res))))))

(defun daytime ()
   (with-client-socket (st "127.0.0.1" "daytime" "tcp")
        (read-line st () `eof)))

(daytime)   =>    "Sunday, May 31, 1998 13:50:35"
```

(with-server-socket (*socket service protocol* **[***ipver***])** *forms****)** -> *<object>* **macro**

This macro creates a new *socket* and connect it to the service *service* using protocol *protocol.* The *ipver* parameter is either **:ipv4** (default) or **:ipv6**. The forms *forms* are executed in order. The last evaluated form is returned and the socket is automatically closed. This macro is very used to talk easily with a client using the standard input/ouput functions.

```
(defmacro with-server-socket (socket &rest body)
   `(let ((,(car socket) (socket)))
         (when ,(car socket)
               (unwind-protect (when ,`(listen ,@socket)
                                     ,@body)
                               (close ,(car socket))))))

(defun simplest-server ()
   ;; Server side (server addr : 193.57.0.1)
   (with-server-socket (server "testtcp" "tcp")
        (let ((fds (create-vector 16 ()))
              (fdr (create-vector 16 ()))
              (fdw (create-vector 16 ())))
             (select-clear fds)
             (select-add server fs)
             (while (eq (select 1 fds fdr fdw () 5.0) 0)
                    (print "Waiting ...."))
             (let ((client (accept server)))
                  ;; Talk with client using standard I/O.
                  (with-standard-input client
                       (with-standard-output client
                            (print (eval (read))))))
             (close client)
             (select-remove server fds))))
```

The following code shoes a complete example of socket interface:

```
;;
;; This code runs on your server
;;

(defglobal *host*     (get-host-address 'coltrane))  ;; server name
(defglobal *service*  'testtcp)                      ;; service name
(defglobal *protocol* 'tcp))                         ;; protocol name

(defglobal *running*  t)

(defun test-stream-server-socket ()
   ;; example to test server in TCP or UDP protocol
   (let ((buf     (create-string 128))
         (socket  ())
```

```
         (client  ())
         (len     0)
         (file1   "src/version.h")
         (file2   "version.h"))
        (setq socket (socket))
        (when (listen socket *service* *protocol*)
              (print "Server   : " (get-host-name))
              (print "Service  : " (get-service-by-name *service*))
              (print "Protocol : " (get-proto-by-name *protocol*))
              (print "Socket   : " socket)
              (print "----------")
              (setq *running* t)
              (while *running*
                     (print "Waiting for connexion ... ")
                     (setq client (accept socket))
                     (print "Socket " client " on "
                            (get-peer-name client))
                     (if (eq *protocol* 'tcp)
                         (setq len (receive client buf 20))
                         (setq len (receive-from client buf 20)))
                     (case-using #'equal (subseq buf 0 len)
                           (("file") ;; Send file to the client
                                   )
                           (("stop") (print "Closed by "
                                                   (get-peer-name client))
                                     (setq *running* ()))
                           (t        (print 'error)))
                     (print "Done with socket " client)
                     (close client))))
```

32 Miscellaneous Functions

`*islisp-version*` — **`constant`**

Returns an integer constant that is the current version of ISLISP standard implemented by OpenLisp. Currently, this variable has the value **`200710`**.

`(system)` -> *<object>* — **`function`**

Always returns the symbol **`openlisp`.**

`(version)` -> *<float>* — **`function`**

Returns current version of **OpenLisp** (always a float).

`(banner)` -> *<boolean>* — **`function`**

Display standard **OpenLisp** banner. This function is only usefull to display the standard welcome message in a core image used with execore *(see also* **`core-init-std`** *function).*

`(pointer-size)` -> *<integer>* — **`function`**

This function returns the size in bytes of a Lisp pointer. This value is 4 on 32 bits machines and 8 for 64 bits machines.

`*print-stat*` -> *<boolean>* — **`dynamic variable`**

When this dynamic variable is set to **t**, the toplevel will print the time spent to evaluate the latest expression as well as the number of GC called for this evaluation.

(gc [**t**]**)** -> *<integer>* **function**

Force a call to the Garbage Collector. If a flag **t** is given, **gc** returns a list that contains the total number of GC for the different object types and the number of free objects for types **<cons>**, **<symbol>**, **<string>**, **<general-vector>, <float>** and **<heap>**. Otherwise, **gc** returns the total number of Garbage Collector made.

(gc-count) -> *<integer>* **function**

Return the number of GC made from the start.

(gc-low-threshold [*ratio*]**)** -> *<float>* **function**

Return (or change if an argument is provided) the GC low memory threshold for which the GC will try to allocate more memory. By defaut, this value is 0.10 (meaning allocate memory when only 10% of this zone remains free after GC).

(gc-growing-factor [*ratio*]**)** -> *<float>* **function**

Return (or change if an argument is provided) the GC growing factor used when more memory is needed (see **gc-low-threshold**). By defaut, this value is 0.33 (meaning allocate 1/3 memory more).

(gc-max-objects [*nbobj*]**)** -> *<integer>* **function**

Return (or change if an argument is provided) the maximum number of objects after which GC will not try to allocate more memory, even if it consider low memory condtion. If this value is 0 (default value), GC will always check if more memory is required.

(gc-min-objects [*nbobj*]**)** -> *<integer>* **function**

Return (or change if an argument is provided) the minimum number of objects that must be free after GC. For each zone, the GC will try to allocate more memory if less than this number of objects are free. By defaut, this value is 1024.

(gc-compact-threshold [*nbobj*]**)** -> *<integer>* **function**

Return (or change if an argument is provided) the filling threshold for heap zone compaction. Value must be in range [0.0 100.0]. The default value is 0.0 meaning always compact. When you change the threshold, it returns the previous value. *Hint: don't try to go far beyond 0.6 or 0.7.*

(gc-free-unused-memory *force-gc keep***)** -> *<object>* **function**

When the implementation supports virtual memory, calling this function will try to reduce the amount of memory already allocated. If *force-gc* is non-**nil**, a GC is made; *keep* is amount of heap to keep free.

(gc-free-unused-memory t 0) *;; will try to free all possible memory from heap*

(gcinfo [*n*]**)** -> *<object>* **function**

Without arguments, **gcinfo** returns the information available after the last Garbage Collector was called. With an argument, it returns the information that was available at startup. This function is generally used to make statistics of memory usage.

(before-gc-hook *n type***)** -> <*object*> **function**
(after-gc-hook *n type***)** -> <*object*> **function**

before-gc-hook is always called before a **gc** is made. It receives *n* the total number of gc made from the beginning and *type* the type of object for which the current gc is called. The same way, **after-gc-hook** is always called with the same arguments after **gc** is made. A user can redefine those two functions but, as a strong advice, **before-gc-hook** should not allocate any memory. Especially, macros the code may contain must be expanded before the first GC is called. You can ensure this by calling manually something like **(before-gc-hook -1 nil)** right after the function is defined.

Example:

```
(defglobal gc-verbose nil)

(defun before-gc-hook (n type)
  (when (and (>= n 0) gc-verbose)
    (format t "BEFORE GC: ~A ~A~%" n type)))

(before-gc-hook -1 nil) ;; ensure all macros are expanded before first GC.
```

(oblist [[*package***]** *access***])** -> <*list*> **function**

Returns the list of all symbols defined in the system. If *package* is given, only symbols in this package are returned. If *access* is **:external** (default **:internal**), only exported symbols in this package are returned.

(character-size) -> <*integer*> **function**

This function returns the size in bytes of a Lisp character. This value is 1 on implementations that support only ASCII codes and 2 on implementations that support UNICODE characters.

(lambda-list-hook <*definition*> <*defname*>**)** -> <*list*> **function**

If defined, this function is called each time a new function, macro of lambda is created. It receives function definition and returns a user modified definition. The second argument is the define function name. It may be used to add additional function behavior like non-standard function keywords, document-string processing or pre-processor optimizations.
(see `contrib/defhook.lsp` example).

33 Hash tables.

A hash table is a Lisp object that can efficiently map a given Lisp object to another Lisp object. Each hash table has a set of *entries*, each of which associates a particular *key* with a particular *value*. The basic functions that deal with hash tables can create entries, delete entries, and find the value that is associated with a given key. Finding the value is very fast, even if there are many entries, because hashing is used; this is an important advantage of hash tables over property lists.
A given hash table can associate only one *value* with a given *key*; if you try to add a second *value*, it will replace the first. Also, adding a value to a hash table is a destructive operation; the hash table is modified. By contrast, association lists can be augmented non-destructively.
Hash tables come in two kinds, the difference being whether the keys are compared with **eq**, **eql**, or **equal**. In other words, there are hash tables that hash on Lisp *objects* (using **eq**) and there are hash tables that hash on *tree structure* (using **equal**).
Hash tables are created with the function **make-hash-table** which takes various options. To look up a key and find the associated value, use **gethash**. New entries are added to hash tables using **setf** with **gethash**. To remove an entry, use **remhash**.

We have the following class relations:

```
<object>
   <hash-table>
```

When a hash table is first created, it has a *size*, which is the maximum number of entries it can hold. Usually the actual capacity of the table is somewhat less, since the hashing is not perfectly collision-free. With the maximum possible bad luck, the capacity could be very much less, but this rarely happens. If so many entries are added that the capacity is exceeded, the hash table will automatically grow, and the entries will be *rehashed* (new hash values will be recomputed, and everything will be rearranged so that the fast hash lookup still works). This is transparent to the caller; it all happens automatically.

(make-hash-table [:size *size*] **[:rehash-threshold** *threshold*] **[:rehash-size** *rsize*] [**:test** *test-function*]**)** -> *<hash-table>* **function**

make-hash-table create This function creates and returns a new hash table. The **:test** argument determines how keys are compared; it must be one of the three values **#'eq**, **#'eql**, or **#'equal**, or one of the three symbols **eq**, **eql**, or **equal**. If no test is specified, **eq** is assumed (*Common Lisp assumes* **eql** *instead).*.

The **:size** argument (a non-negative integer) sets the initial size of the hash table, in entries. (The actual size may be rounded up from the size you specify to the next "good" size, for example to make it a prime number.). You won't necessarily be able to store precisely this many entries into the table before it overflows and becomes bigger, but this argument does serve as a hint to the implementation of approximately how many entries you intend to store.

The **:rehash-size** argument specifies how much to increase the size of the hash table when it becomes full. This can be an integer greater than zero, which is the number of entries to add, or it can be a floating-point number greater than 1, which is the ratio of the new size to the old size. The default value for this argument is 1.5.

The **:rehash-threshold** argument specifies how full the hash table can get before it must grow. It may be any **real** number between **0** and **1**, inclusive. It indicates the maximum desired level of hash table occupancy. An implementation is permitted to ignore this argument. The default value for this argument is 0.75.

(hash-table-p *hash-table***)** -> *<object>* **function**

hash-table-p is true if its argument is a hash table, and otherwise is false.

(hash-table-count *hash-table***)** -> *<integer>* **function**

This returns the number of entries in the *hash-table*. When a hash table is first created or has been cleared, the number of entries is zero.

(hash-table-test *hash-table***)** -> *<object>* **function**

hash-table-test returns the test function (as symbol) of a hash table.

(hash-table-size *hash-table***)** -> *<integer>* **function**

hash-table-size returns the current size of a hash table.

(hash-table-rehash-size *hash-table***)** -> *<object>* **function**

hash-table-rehash-size returns the current rehash size.

(hash-table-rehash-threshold *hash-table***)** -> *<object>* **function**

hash-table-rehash-threshold returns the current rehash threshold.

(gethash *key hash-table* **[** *default* **])** -> <*object*> **function**

gethash finds the entry in *hash-table* whose key is *key* and returns the associated value. If there is no such entry, **gethash** returns *default*, which is **nil** if not specified.
gethash actually returns two values, the second being a predicate value that is true if an entry was found, and false if no entry was found.
setf may be used with **gethash** to make new entries in a hash table. If an entry with the specified *key* already exists, it is removed before the new entry is added. The *default* argument may be specified to **gethash** in this context; it is ignored by **setf** but may be useful in such macros as **incf** that are related to **setf**:

```
(incf (gethash a-key table 0))
```

means approximately the same as

```
(setf (gethash a-key table 0) (+ (gethash a-key table 0) 1))
```

which in turn would be treated as simply

```
(setf (gethash a-key table) (+ (gethash a-key table 0) 1))
```

(puthash *key hash-table value***)** -> <*object*> **function**
(setf (gethash *key hash-table* **[** *default* **])** *value***)** -> <*object*> **function**

setf may be used with **gethash** to make new entries in a hash table using puthash. If an entry with the specified *key* already exists, it is removed before the new entry is added. The *default* argument may be specified to **gethash** in this context; it is ignored by **setf** but may be useful in such macros as **incf** that are related to **setf.**

(remhash *key hash-table***)** -> <*object*> **function**

remhash removes any entry for *key* in *hash-table*. This is a predicate that is true if there was an entry or false if there was not.

(clrhash *hash-table***)** -> <*object*> **function**

This removes all the entries from *hash-table* and returns the hash table itself.

(rehash *hash-table count***)** -> <*object*> **function**

Rebuilts *hash-table* with a number of keys equals to *count*.

(sxhash *object* **[** *n* **])** -> <*object*> **function**

sxhash computes a hash code for an object and returns the hash code as a non-negative fixnum. A property of **sxhash** is that **(equal** *x y***)** implies **(= (sxhash** *x***) (sxhash** *y***))**.
The manner in which the hash code is computed is implementation-dependent but independent of the particular ``incarnation'' or ``core image.'' Hash values produced by **sxhash** may be written out to files, for example, and meaningfully read in again into an instance of the same implementation. The function **sxhash** is a convenient tool for the user who needs to create more complicated hashed data structures than are provided by **hash-table** objects.

(hash *object* **[** *n* **])** -> <*object*> **function**

hash computes a hash code for a symbol and returns the hash code as a non-negative fixnum. A property of **hash** is that **(equal** *x y***)** implies **(= (hash** *x***) (hash** *y***))**.

(maphash *function hash-table***)** -> <*null*> **function**

For each entry in *hash-table*, **maphash** calls *function* on two arguments: the key of the entry and the value of the entry; **maphash** then returns **nil**. If entries are added to or deleted from the hash table while a **maphash** is in progress, the results are unpredictable, with one exception: if the *function* calls **remhash** to remove the entry currently being processed by the *function*, or performs a **setf** of **gethash** on that entry to change the associated value, then those operations will have the intended effect.

Example:

```
;;; Alter every entry in MY-HASH-TABLE, replacing the value with
;;; its square root.  Entries with negative values are removed.
(maphash #'(lambda (key val)
             (if (minusp val)
                 (remhash key my-hash-table)
                 (setf (gethash key my-hash-table) (sqrt val))))
         my-hash-table)
```

34 Regular expressions.

OpenLisp has optional module that implements regular expression search. A regular expression is an object of type **<regexp>** that has been compiled from a pattern string using **regcomp** function. This regular expression is then matched against a string (or a symbol) using **regexe** function. If a match is found, **regexe** returns **t.** If a match is found and a vector of 2 elements is passed as a third argument it is filled with the indexes that match the expression. A simpler function, **regmatch**, is simply the combination of **regcomp** and **regexe** in only one call.

A **regular expression** is zero or more branches, separated by '|'. It matches anything that matches one of the branches.

A **branch** is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A **piece** is an atom possibly followed by '*', '+', or '?'. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a match of the atom, or the null string.

An **atom** is a regular expression in parentheses (matching a match for the regular expression), a range (see below), '.' (matching any single character), '^' (matching the null string at the beginning of the input string), '$' (matching the null string at the end of the input string), '\\' followed by a single character (matching that character), or a single character with no other significance (matching that character).

A **range** is a sequence of characters enclosed in '[]'. It normally matches any single character from the sequence. If the sequence begins with '^', it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by '\-', this is shorthand for the full list of ASCII characters between them (e.g. '[0-9]' matches any decimal digit). To include a literal ']' in the sequence, make it the first character (following a possible '^'). To include a literal '\-', make it the first or last character.

(regexp-p *object***)** -> *<boolean>* **function**

Returns **t** if *object* is a computed regular expression (of type **<regexp>**), **nil** otherwise.

(regcomp *regular-expression***)** -> *<regexp>* **function**

Compute *regular-expression* in an internal format and returns a **<regexp>** object.

(regexe *regexp string retvect* [*start*]**)** -> *<regexp>* **function**

Match *string* against *regexp* starting at *start* position (default 0). It returns **t** if expression string match *expression* and **nil** otherwise. The third argument *retvect* can be **nil** or a vector of two elements. In that case, it is filled with the included lower and excluded upper indexes of the match.

Example:

```
(defglobal x (regcomp "[A-Z]*"))    => x
(regexp-p x)                        => t
(regexe x "ABCAB" nil)              => t
(regexe x "abcab" nil)              => nil
(defglobal y #(0 0))                => y
(regexe x "ABCAB" y)                => t
y                                   => #(0 5)
(regexe x "ABCAB" y 1)              => t
y                                   => #(1 5)
```

(rematch *string1 string2* [*start*]**)** -> <*boolean*> **function**

Match *string1* against *string2* starting at *start* position (default 0). It returns **t** if expression string *string2* match the regular expression *string1* and **nil** otherwise.

```
(regmatch "^[A-Z]oo" "Foo Bar")     => t
(regmatch "^[A-Z]" "Foo Bar" 2)     => nil
(regmatch "^[A-Z]" "Foo Bar" 4)     => t
```

35 Windows registry.

On most Windows ports, **OpenLisp** has an optional registry module to manage keys and values stored in Windows registry. The registry is a system-defined database in which applications and system components store and retrieve configuration data. The data stored in the registry varies according to the version of Microsoft Windows. Applications use the registry API to retrieve, modify, or delete registry data.
The system defines predefined keys that are always open. Predefined keys help an application navigate in the registry and make it possible to develop tools that allow a system administrator to manipulate categories of data. OpenLisp defines the following symbols for those predefined keys:

HKEY_CLASSES_ROOT	Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.
HKEY_CURRENT_CONFIG	Contains information about the current hardware profile of the local computer system. The information under HKEY_CURRENT_CONFIG describes only the differences between the current hardware configuration and the standard configuration. Information about the standard hardware configuration is stored under the Software and System keys of HKEY_LOCAL_MACHINE.
HKEY_LOCAL_MACHINE	Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software. It contains subkeys that hold current configuration data, including Plug and Play information, network logon preferences, network security information, software-related information (such as server names and the location of the server), and other system information.
HKEY_USERS	Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

(reg-create-key <*key*> <*subkey*> <*name*>**)** -> <*object*> **function**

This function creates the specified registry key. If the key already exists in the registry, the function opens it.

(reg-delete-key <*key*> <*subkey*> <*name*>**)** -> <*object*> **function**

This function deletes a subkey, including all of its values.

(reg-enum-key *<key> <subkey>***)** -> *<object>* **function**

This function returns a list of subkeys for the given key.

(reg-get-value *<key> <subkey> <name>***)** -> *<object>* **function**

This function returns the value associated with <name> in the subkey entry.

(reg-set-value *<key> <subkey> <name> <value>***)** -> *<object>* **function**

This function set (or change) the value associated with <name> in the subkey entry.

(reg-rem-value *<key> <subkey> <name>***)** -> *<object>* **function**

This function removes the value associated with <name> in the subkey entry.

(reg-get-type *<key> <subkey> <name>***)** -> *<integer>* **function**

This function returns an integer value that represents the value type assosciated to <name>.

(reg-get-typename *<key> <subkey> <name>***)** -> *<string>* **function**

This function returns a new string that represents the value type name assosciated *<name>*.

(reg-enum-value *<key> <subkey>***)** -> *<object>* **function**

This function returns a list of entries strings for the given subkey.

36 OpenLisp Compiler

OpenLisp has a compiler that translates Lisp expressions into compiled equivalent form. The code is faster and less expensive in memory.

37 Compiler variables and functions.

`*compile-verbose*` -> *<integer>* **`variable`**

This dynamic variable provides the default for the **`:verbose`** argument to **`compile-file`**. Its initial value is 0.

`(compile` *name* [*definition*]**`)`** -> *<function>* **`function`**

If *definition* is supplied, it should be a lambda-expression, the interpreted function to be compiled. If it is not supplied, then *name* should be a symbol with a definition that is a lambda-expression; that definition is compiled and the resulting compiled code is put back into the symbol as its function definition.

`(compile-file` *infile* **`[:output-file`** *outfile*] **`[:verbose`** *flag*] **`[:print`** *flag*] **`[:cdata`** *flag*]**`)`**
-> *<function>* **`function`**

The *infile* should be a Lisp source file; its contents are compiled and written as a binary object file.

The **`:output-file`** argument may be used to specify an output pathname; it defaults in a manner appropriate to the implementation's file system conventions. By default, compiled files have **`.lap`** suffix.

The **`:verbose`** argument (which defaults to the value of dynamic variable **`*compile-verbose*`**), if true, permits **`compile-file`** to print a message in the form of a comment to **`(standard-output)`** indicating what file is being compiled and other useful information.

The **`:print`** argument (which defaults to the value of **`*compile-print*`**), if true, causes information about top-level forms in the file being compiled to be printed **`(standard-output)`**.

The **:cc** argument (which defaults to the value of ***compile-to-c***) permits **compile-file** to generate C file instead of lap file.

The **:cdata** argument (which defaults to the value of ***compile-to-c***) permits **compile-file** to generate C file instead of lap file. It assumes that the file mainly contains data. The generated code is generally smaller than with **:cc** option but it is suitable only for small data (i.e. 2 or 3 lines). It may also hang some compilers that limit constant strings.

(disassemble *name***)** -> *<null>* **function**

The argument should be a function object, a lambda-expression, or a symbol with a function definition. If the relevant function is not a compiled function, it is first compiled. In any case, the compiled code is then “reverse-assembled” and printed out in a symbolic format. When **disassemble** compiles a function, it never installs the resulting compiled-function object in the **symbol-function** of a symbol. This is primarily useful for debugging the compiler.

38 Standalone applications

As C or FORTRAN compilers, **OpenLisp** (unlike most other Lisp systems) can produce standalone executables. You can also make a new library from your code that you can embed with standard **OpenLisp** libries in a C or C++ program. For that, you must have the same C/C++ compiler used to create **OpenLisp** system.

39 General principle.

Here are the required steps to compile your own Lisp code into a standalone executable:

- compile Lisp files to C (foo.lsp -> foo.c)
- compile the generated C files to machine code object (foo.c -> foo.obj *or foo.o*)
- link the object files with Lisp kernel and Lisp standard libraries (foo.obj -> foo.exe *or foo*)

A new application is created by creating a new copy of an existing application directory and naming it app (where app is the application name). The basic `http` application is useful as a template for creating new applications.

40 Create your own project

To help you, **OpenLisp** contains some projects with associated Makefiles that show you how to achieve this goal. For example, the `http` directory contains a little project for a simple HTTP server entirely written in Lisp.

To make your own project named "`app`":

- make a new directory `openlisp-X.Y.0/app`
- change directory to `openlisp-X.Y.0/app`
- copy `Makefile` file from `openlisp-X.Y.0/http`
- edit `Makefile` to add your own files and set `MODDIR` to your source directory '".". If the files are located in the same "`app`" directory.
- call "`make compile`" to compile in C your lisp files.
- call "`make`" to actually build a new library that contains your compile application and a new executable that contains a strandard **OpenLisp** environment and your compile function ready to use.

The Microsoft VC++ Makefile looks like:

```
#
#     Makefile for LAP->C generated files (c) C. Jullien 2005/06/11
#
# Module name and files.
#

MODNAME          = http
MODDIR           = .
MODGEN           = .
MODREG           = utf8 cgi httpd
MODCPFLAGS       = :cc
MODCCOPT         = -DOLLAPOPTIMIZE
MODLIST          = httpserv.obj
MODOPT           = -D_ODSP -D_NOBANNER -D_USEROPTIONS

!if "$(OPENLISP)" == ""
!include "../src/common.mak"
!include "../src/module.mak"
!else
!include "$(OPENLISP)/src/common.mak"
!include "$(OPENLISP)/src/module.mak"
!endif
```

Where:

MODNAME	is the name of binary (`http.exe`)
MODREG	is the list a standard library module to register (`utf8`, `cgi` and `httpd`)
MODLIST	is the list of Lisp file to be compiled for this project (*i.e.* Lisp filename with no extension).
MODOPT	is the list of optional flags to pass to the C compile (hide banner, allow user options)
MODCCOPT	is the flag to change compile code behavior (see below).
MODCPFLAG	is an optional Lisp compiler flag `:data` means that Lisp code mainly contains data.
MODDIR	is the relative path of source files. "." if the Lisp source file are in the same directory.

To generate your own standalone executable you just have to run:

```
$ make compile && make              # unix syntax
C:> nmake compile && nmake          # Windows syntax
```

41 Tuning and debugging your application

There are some usefull flags you can add in `MODCCOPT` variable to change the behaviour of your compiled code. Please note that those flags are mutually exclusive.

OLLAPOPTIMIZE	This flag produce the fastest code but does less checking. Use it only when your application is fully tested.
OLLAPTRACE	This flag add an optional module trace that you can dynamically activate when an environment variable called `OLCHECKMODULE` is set. A simple message is shown when each compiled module is initialized. It's usefull to detect which module is loaded.
OLLAPTIMETRACE	When this flag is set, each compile function is automatically profiled. At any time you can call one of the profile functions (see profiling section) to see the number of calls and the time spent in a function.

42 Executable core image

You can create another kind of executable using a feature named ‘execore’. An execore just combines the **OpenLisp** binary file and a core image in a single file that you can call as any other standard executable file.

To create and execore, you must create a core image using **`save-core`** function (see related section). Then, using the execore shell script, you can combine this core and **OpenLisp** with the following command:

```
$ execore uxlisp mycore.cor mybin                  # unix syntax
C:>execore openlisp.exe mycore.cor mybin.exe       # Windows syntax
```

43 Standard Library

44 Module and Package Functions.

A *module* is a Lisp subsystem that is loaded from one or more files. A module is normally loaded as a single unit, regardless of how many files are involved. A module may consist of one package or several packages. The file-loading process is necessarily implementation-dependent and **OpenLisp** provides some very simple portable machinery for naming modules, for keeping track of which modules have been loaded, and for loading modules as a unit. Features can be used with conditionnal reader forms **#+** et **#-.**

modules -> *<list>* **dynamic variable**

The variable ***modules*** is a list of names of the modules that have been loaded into the Lisp system so far. This list is used by the functions **provide** and **require**.

(require *module-name* [*pathname**]**)** -> *<object>* **function**
(provide *module-name***)** -> *<object>* **function**

Each module has a unique name (a string). The **provide** and **require** functions accept either a string or a symbol as the *module-name* argument. If a symbol is provided, its print name is used as the module name. If the module consists of a single package, it is customary for the package and module names to be the same.
The **provide** function adds a new module name to the list of modules maintained in the variable ***modules***, thereby indicating that the module in question has been loaded.
The **require** function tests whether a module is already present (using a case-sensitive comparison); if the module is not present, **require** proceeds to load the appropriate file or set of files. The *pathname* argument, if present, is a single pathname or a list of pathnames whose files are to be loaded in order, left to right. If the *pathname* argument is **nil** or is not provided, the system will attempt to determine, in some system-dependent manner, which files to load. This will typically involve some central registry of module names and the associated file lists.

(featurep *feature***)** -> *<object>* **function**

Test if the feature *feature* is present in the system.

package -> *<symbol>* **dynamic variable**

The value of this dynamic variable must be a symbol; this symbol is said to be the current package. The initial value of ***package*** is the **user** package.

(packagep *package***)** -> *<symbol>* **function**

Returns **t** if *package* is a package objetc type, **nil** otherwise.

(create-package *package-name***)** -> *<symbol>* **dynamic variable**

This creates and returns a new package with the specified package name. As described above, this argument may be either a string or a symbol.

(find-package *name***)** -> *<symbol>* **function**

The *name* must be a string that is the name for a package. This argument may also be a symbol, in which case the symbol's print name is used. The package with that name is returned; if no such package exists, **find-package** returns **nil**. The matching of names observes case (as in **string=**).

(package-name *package***)** -> *<symbol>* **function**

The argument must be a package. This function returns the string that names that package. The *package* argument may be either a package object or a package name

(package-nickames *package***)** -> *<symbol>* **function**

The argument must be a package. This function returns the list of nickname strings for that package, not including the primary name.

(package-use-list *package***)** -> *<symbol>* **function**

A list of other packages used by the argument package is returned. The *package* argument may be either a package object or a package name.

(package-used-by-list *package***)** -> *<symbol>* **function**

A list of other packages that use the argument *package* is returned. The *package* argument may be either a package object or a package name.

(package-shadowing-symbols *package***)** -> *<symbol>* **function**

A list is returned of symbols that have been declared as shadowing symbols in this package by **shadow**. All symbols on this list are present in the specified package. The *package* argument may be either a package object or a package name.

(list-all-packages) -> *<symbol>* **function**

This function returns a list of all packages that currently exist in the Lisp system.

(delete-package *package***)** -> *<symbol>* **function**

The **delete-package** function deletes the specified *package* from all package system data structures. The *package* argument may be either a package or the name of a package.

(in-package *package-name***)** -> <*symbol*> **function**

The **in-package** function is intended to be placed at the start of a file containing a subsystem that is to be loaded into some package other than **user**.
This function sets ***package*** to is set to *package-name*. This binding will remain in force until changed by the user (perhaps with another **in-package** call) or until the ***package*** variable reverts to its old value at the completion of a **load** operation.

(use-package *packages-to-use* [*package*]**)** -> <*symbol*> **function**

The *packages-to-use* argument should be a list of package names, or possibly a single package or package name. These packages are added to the use-list of *package* if they are not there already. All external symbols in the packages to use become accessible in *package* as internal symbols. It is an error to try to use the **keyword** package. **use-package** returns the previous package.

(export *symbols* [*package*]**)** -> <*symbol*> **function**

The *symbols* argument should be a list of symbols, or possibly a single symbol. These symbols become accessible as external symbols in package *package*. **export** returns **t**.
By convention, a call to **export** listing all exported symbols is placed near the start of a file to advertise which of the symbols mentioned in the file are intended to be used by other programs.

(import *symbols* [*package*]**)** -> <*symbol*> **function**

The argument should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in package *package* and can therefore be referred to without having to use qualified-name (colon) syntax. **import** signals a correctable error if any of the imported symbols has the same name as some distinct symbol already accessible in the package. **import** returns **t**.

(shadowing-import *symbols* [*package*]**)** -> <*symbol*> **function**

This is like **import**, but it does not signal an error even if the importation of a symbol would shadow some symbol already accessible in the package. In addition to being imported, the symbol is placed on the shadowing-symbols list of package. **shadowing-import** returns **t**.
shadowing-import should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change.

(shadow *symbols* [*package*]**)** -> <*symbol*> **function**

The argument should be a list of symbols, or possibly a single symbol. The print name of each symbol is extracted, and the specified *package* is searched for a symbol of that name. If such a symbol is present in this package, then nothing is done. Otherwise, a new symbol is created with this print name, and it is inserted in the *package* as an internal symbol. The symbol is also placed on the shadowing-symbols list of the *package*. **shadow** returns **t**.

(intern *string* [*package*]**)** -> <*symbol*> **function**

The *package,* which defaults to the current package, is searched for a symbol with the name specified by the *string* argument. If a symbol with the specified name is found, it is returned. If no such symbol is found, one is created and is installed in the specified package as an internal symbol (as an external symbol if the package is the **keyword** package); the specified package becomes the home package of the created symbol.

(unintern *symbol* [*package*]**)** -> <*symbol*> **function**

If the specified symbol is present in the specified *package*, it is removed from that package and also from the package's shadowing-symbols list if it is present there. Moreover, if the *package* is the home package for the symbol, the symbol is made to have no home package. Note that in some circumstances the symbol may continue to be accessible in the specified package by inheritance. **`unintern`** returns **`t`** if it actually removed a symbol, and **`nil`** otherwise.

`(find-all-symbols` *symbol-name***`)`** -> <*symbol*> **`function`**

`find-all-symbols` searches every package in the system to find every symbol whose print name is the specified string. A list of all such symbols found is returned. This search is case-sensitive. If the argument is a symbol, its print name supplies the string to be searched for.

`(defpackage` *defined-package-name* {*option*}***`)`** -> <*symbol*> **`macro`**

This creates a new package, or modifies an existing one, whose name is *defined-package-name*. The *defined-package-name* may be a string or a symbol; if it is a symbol, only its print name matters, and not what package, if any, the symbol happens to be in. The newly created or modified package is returned as the value of the **`defpackage`** form.
Each standard *option* is a list of a keyword (the name of the option) and associated arguments. No part of a **`defpackage`** form is evaluated. More than one option of the same kind may occur within the same **`defpackage`** form.
The standard options for **`defpackage`** are as follows. In every case, any option argument called *package-name* or *symbol-name* may be a string or a symbol; if it is a symbol, only its print name matters, and not what package, if any, the symbol happens to be in.

`(`**`:import`** {*`symbol-name`*}*`)`

> Symbols with the specified names are located in the global package. These symbols are imported into the package being defined, just as with the function **`import`**.

`(`**`:nicknames`** *`{package-name}`*`*)`

> The specified names become nicknames of the package being defined.

`(`**`:import-from`** *`package-name`* {*`symbol-name`*}*`)`

> Symbols with the specified names are located in the specified package. These symbols are imported into the package being defined, just as with the function **`import`**.

`(`**`:export`** {*`symbol-name`*}*`)`

> Symbols with the specified names are located or created in the package being defined and then exported, just as with the function **`export`**.

`(`**`:shadow`** *`{symbol-name}`*`*)`

> Symbols with the specified names are created as shadows in the package being defined, just as with the function **`shadow`**.

`(`**`:use`** {*`package-name`*}*`)`

> The package being defined is made to ``use'' the packages specified by this option, just as with the function **`use-package`**. If no **`:use`** option is supplied, then a default list is assumed.

The order in which options appear in a **`defpackage`** form does not matter; part of the convenience of **`defpackage`** is that it sorts out the options into the correct order for processing. Options are processed in the following order:

`:shadow`
`:use`
`:import`

:import-from
:export

If no package named *defined-package-name* already exists, **defpackage** creates it. If such a package does already exist, then no new package is created. The existing package is modified, if possible, to reflect the new definition. The results are undefined if the new definition is not consistent with the current state of the package.
An error is signaled if the same *symbol-name* argument (in the sense of comparing names with **string=**) appears more than once among the arguments to all the **:shadow**, **:import**, **:import-from**, and **:intern** options.
An error is signaled if the same *symbol-name* argument (in the sense of comparing names with **string=**) appears more than once among the arguments to all the **:intern** and **:export** options.
Other kinds of name conflicts are handled in the same manner that the underlying operations **import**, and **export** would handle them.

45 Defining Structures.

OpenLisp provides a facility for creating named record structures with named components. In effect, the user can define a new data type; every data structure of that type has components with specified names. Constructor, access, and assignment constructs are automatically defined when the data type is defined
All structures are defined through the **defstruct** construct. A call to **defstruct** defines a new data type whose instances have named slots. Structures are generally faster than object created by **defclass.**
Two syntaxes, Common Lisp that is the default and Le-Lisp (not described there), can be used as the same time. Only the Common Lisp syntax allows inheritance. Slots use an optimized internal mechanism faster than vector access. As a general advice, structure names should be defined with enclosed **< >** character since a structure can be seen as a new type. Those two extra characters are not taken to generate access functions. That way, the structure named **<foo>** wil generate **make-foo**, **foo-p**, … instead of **make-<foo>** or **<foo>-p**,…

structure-standard-names	**dynamic variable**

This dynamic variable is set to **t** when Common Lisp compatibility is in use (default). To generate structure names in the same way as Le-Lisp, you must set it to **nil**.

defstruct -> <*symbol*>	**feature**

defstruct feature is present when the defstruct functions have been loaded in the System.

(defstruct *struct* **[***slot-description***]*)** -> <*object*>	**macro**

This defines a record-structure data type. A general call to **defstruct** looks like the following example.

```
(defstruct (name option-1 option-2 ... option-m)
           slot-description-1
           slot-description-2
           ...
           slot-description-n)
```

The *name* must be a symbol; it becomes the name of a new data type consisting of all instances of the structure. The function **instancep** will accept and use this name as appropriate. The *name* is returned as the value of the *defstruct* form.
Usually no options are needed at all. If no options are specified, then one may write simply *name* instead of **(***name***)** after the word **defstruct**.

Each *slot-description-j* is of the form
(*slot-name default-init slot-option-name-1 slot-option-value-1 slot-option-name-2 slot-option-value-2* **...** *slot-option-name-kj slot-option-value-kj***)**
Each *slot-name* must be a symbol; an access function is defined for each slot. If no options and no *default-init* are specified, then one may write simply *slot-name* instead of **(***slot-name***)** as the slot description.

The *default-init* is a form that is evaluated *each time* a structure is to be constructed; the value is used as the initial value of the slot.
If no *default-init* is specified, then the initial contents of the slot are undefined and implementation-dependent.
defstruct not only defines an access function for each slot, but also arranges for **setf** to work properly on such access functions, defines a predicate named *name*-**p**, defines a constructor function named **make-***name*, and defines a copier function named **copy-***name*.

Note: when the name is surrounded with **<** and **>**, those two characters are not used for generated names.

Each *slot-description* in a **defstruct** form may specify one or more slot-options. A slot-option consists of a pair of a keyword and a value (which is not a form to be evaluated, but the value itself).

Example:

```
(defstruct <ship>
   (x-position 0.0)
   (y-position 0.0)
   (x-velocity 0.0)
   (y-velocity 0.0)
   (mass *default-ship-mass*))
```

The only slot-option is:

:read-only

The option **:read-only** *x*, where *x* is not **nil**, specifies that this slot may not be altered; it will always contain the value specified at construction time. **setf** will not accept the access function for this slot. If *x* is **nil**, this slot-option has no effect. Note that the argument form *x* is not evaluated.

Note that it is impossible to specify a slot-option unless a default value is specified first.

:conc-name

This provides for automatic prefixing of names of access functions. It is conventional to begin the names of all the access functions of a structure with a specific prefix, the name of the structure followed by a hyphen. This is the default behavior.

The argument to the **:conc-name** option specifies an alternative prefix to be used. (If a hyphen is to be used as a separator, it must be specified as part of the prefix.) If **nil** is specified as an argument, then *no* prefix is used; then the names of the access functions are the same as the slot-names, and it is up to the user to name the slots reasonably.

Note that no matter what is specified for **:conc-name**, with a constructor function one uses slot keywords that match the slot-names, with no prefix attached. On the other hand, one uses the access-function name when using **setf**.

Example:

```
(defstruct <door> knob-color width material)
(setq my-door
      (make-door :knob-color 'red :width 5.0))
(door-width my-door) => 5.0
(setf (door-width my-door) 43.7)
(door-width my-door) => 43.7
(door-knob-color my-door) => red
```

:constructor

This option takes one argument, a symbol, which specifies the name of the constructor function. If the argument is not provided or if the option itself is not provided, the name of the constructor is produced by concatenating the string **"make-"** and the name of the structure. If the argument is provided and is **nil**, no constructor function is defined.

:copier

This option takes one argument, a symbol, which specifies the name of the copier function. If the argument is not provided or if the option itself is not provided, the name of the copier is produced by concatenating the string **"copy-"** and the name of the structure, putting the name in whatever package is current at the time the **defstruct** form is processed. If the argument is provided and is **nil**, no copier function is defined.

The automatically defined copier function simply makes a new structure and transfers all components verbatim from the argument into the newly created structure. No attempt is made to make copies of the components. Corresponding components of the old and new structures will therefore be **eql**.

:predicate

This option takes one argument, which specifies the name of the type predicate. If the argument is not provided or if the option itself is not provided, the name of the predicate is made by concatenating the name of the structure to the string **"-P"**. If the argument is provided and is **nil**, no predicate is defined. A predicate can be defined only if the structure is ``named"

:include

This option is used for building a new structure definition as an extension of an old structure definition. As an example, suppose you have a structure called **<person>** that looks like this:

```
(defstruct <person> name age sex)
```

Now suppose you want to make a new structure to represent an astronaut. Since astronauts are people too, you would like them also to have the attributes of **name**, **age**, and **sex**, and you would like Lisp functions that operate on **person** structures to operate just as well on **astronaut** structures. You can do this by defining **astronaut** with the **:include** option, as follows:

```
(defstruct (<astronaut> (:include <person>)
                        (:conc-name astro-))
    helmet-size
    (favorite-beverage 'tang))
```

The **:include** option causes the structure being defined to have the same slots as the included structure. This is done in such a way that the access functions for the included structure will also work on the structure being defined. In this example, an **astronaut** will therefore have five slots: the three defined in **person** and the two defined in **astronaut** itself. The access functions defined by the **person** structure can be applied to instances of the **astronaut** structure, and they will work correctly. Moreover, **astronaut** will have its own access functions for components defined by the **person** structure. The following examples illustrate how you can use **astronaut** structures:

```
(setq x (make-astronaut :name 'buzz
                        :age 45
                        :sex t
                        :helmet-size 17.5))

(person-name x)                  => buzz
(astro-name x)                   => buzz
(astro-favorite-beverage x)      => tang
```

The difference between the access functions **person-name** and **astro-name** is that **person-name** may be correctly applied to any **person**, including an **astronaut**, while **astro-name** may be correctly applied only to an **astronaut**. (An implementation may or may not check for incorrect use of access functions.)

At most one **:include** option may be specified in a single **defstruct** form. The argument to the **:include** option is required and must be the name of some previously defined structure.

The structure name of the including structure definition becomes the name of a data type, of course, and therefore a valid type specifier recognizable by **instancep**; moreover, it becomes a subtype of the included structure. In the above example, **<astronaut>** is a subtype of **<person>**; hence

```
(instancep (make-astronaut) (class <person>))
```

is true, indicating that all operations on persons will also work on astronauts.

The following is an advanced feature of the **:include** option. Sometimes, when one structure includes another, the default values or slot-options for the slots that came from the included structure are not what you want. The new structure can specify default values or slot-options for the included slots different from those the included structure specifies, by giving the **:include** option as

(:include *name slot-description-1 slot-description-2 ...***)**

Each *slot-description-j* must have a *slot-name* or *slot-keyword* that is the same as that of some slot in the included structure. If *slot-description-j* has no *default-init*, then in the new structure the slot will have no initial value. Otherwise its initial value form will be replaced by the *default-init* in *slot-description-j*. A normally writable slot may be made read-only. If a slot is read-only in the included structure, then it must also be so in the including structure. If a type is specified for a slot, it must be the same as, or a subtype of, the type specified in the included structure. If it is a strict subtype, the implementation may or may not choose to error-check assignments.

For example, if we had wanted to define **astronaut** so that the default age for an astronaut is **45**, then we could have said:

```
(defstruct (<astronaut> (:include <person> (age 45)))
    helmet-size
    (favorite-beverage 'tang))
```

:print-function

The argument to the **:print-function** option should be a function of three arguments, in a form acceptable to the **function** special form, to be used to print structures of this type. When a structure of this type is to be printed, the function is called on three arguments: the structure to be printed, a stream to print to, and an integer indicating the current depth (to be compared against `*print-level*`). If the **:print-function** option is not specified then a default printing function is provided for the structure that will print out all its slots using **#S** syntax.

(structurep *object***)** -> *<object>* **macro**

structurep returns **t** if object is an instance of structure (*i.e.* if *object* is an instance of **<standard-structure>** class).

46 Sorting.

These functions may destructively modify argument sequences in order to put a sequence into sorted order or to merge two already sorted sequences.

sort -> *<symbol>* **feature**

sort feature is present when the sort functions have been loaded in the System.

(sort *sequence predicate***)** -> *<object>* **function**

The *sequence* is destructively sorted according to an order determined by the *predicate*. The *predicate* should take two arguments, and return non-**nil** if and only if the first argument is strictly less than the second (in some appropriate sense). If the first argument is greater than or equal to the second (in the appropriate sense), then the *predicate* should return **nil**.
The **sort** function determines the relationship between two elements by giving keys extracted from the elements to the *predicate*.
The sorting operation performed by **sort** is not guaranteed *stable*. Elements considered equal by the *predicate* may or may not stay in their original order. (The *predicate* is assumed to consider two elements *x* and *y* to be equal if **(funcall** *predicate x y***)** and **(funcall** *predicate y x***)** are both false.).

The sorting operation may be destructive in all cases. In the case of an array argument, this is accomplished by permuting the elements in place. In the case of a list, the list is destructively reordered in the same manner as for **nreverse**. Thus if the argument should not be destroyed, the user must sort a copy of the argument.

47 Date Library.

Date library can be used to retrieve date informations from the unformatted vector (of type **<date>**) returned by **get-internal-date** function. This feature is Y2C compliant.

date **feature**

date feature is present when the date functions have been loaded in the System.

(get-internal-date *date tz-flag***)** -> *<date>* **function**

Returns a **<date>** object. If *date* is already a **<date>** object (as returned by **make-date**), the structure filled with the values of current date, when set to **nil**, a new **<date>** structure is allocated. If *tz-flag* is **:localtime**, the returned values are for the computer current local time. When *tz-flag* is **:gmt**, the returned values are for GMT.

(date-month *date* [*value*]**)** -> *<object>* **function**
(date-day *date* [*value*]**)** -> *<object>* **function**
(date-year *date* [*value*]**)** -> *<object>* **function**
(date-hour *date* [*value*]**)** -> *<object>* **function**
(date-min *date* [*value*]**)** -> *<object>* **function**
(date-sec *date* [*value*]**)** -> *<object>* **function**
(date-week-day *date* [*value*]**)** -> *<object>* **function**
(date-year-day *date* [*value*]**)** -> *<object>* **function**
(date-daylight-saving *date* [*value*]**)** -> *<object>* **function**
(date-tz *date* [*value*]**)** -> *<object>* **function**
(date-time *date* [*value*]**)** -> *<object>* **function**

With one argument, those functions return (**month**, **day**, **year**, **hour**, **min**, **sec**, **weekday**, **year-day**, **daylight-saving**, **tz** and **time**) from a date object *date.* The optional second argument may be used to alter the value. **year-day** and **daylight-saving** are reserved for future use.

(date-p *date***)** -> *<boolean>* **function**

Returns **t** if *date* is a date object or **nil** otherwise.

(date-difference *date1 date2***)** -> *<integer>* **function**

Returns the difference in seconds of two **date** objects.

(date= *date1 date2***)** -> *<boolean>* **function**
(date/= *date1 date2***)** -> *<boolean>* **function**
(date> *date1 date2***)** -> *<boolean>* **function**
(date>= *date1 date2***)** -> *<boolean>* **function**
(date< *date1 date2***)** -> *<boolean>* **function**
(date<= *date1 date2***)** -> *<boolean>* **function**

Compare two dates.

(date-string *date***)** -> *<string>* **function**

Returns a new human readable string with date information extracted from date object *date*.

Example:

```
(date-string (get-internal-date nil :localtime))
      ⇨ "Sun Nov 22 1998 - 11:13:09"
```

(date-to-time *date***)** -> *<integer>* **function**

Returns an integer which is the number of seconds from system EPOCH.

(time-to-date *time***)** -> *<date>* **function**

Returns a date from *time* which is the number of seconds from system EPOCH. If *time* is 0, it returns the current GMT date.

48 FASL (*obsolete*) Format.

FASL (FASt Load) is an **obsolete** compressed format that can be used to reduce the time spent to read a file. Specially, symbols are only read once and the then only references to those symbols are used on all other places in the file. The sharp **#&***nnn* syntax is use to make a reference to *nnn*$^{\text{nth}}$ element of internal fasl table. You can expect from 50 to more 100 % saving in both size and load time for that file. As convention, fasl files are placed in fasl directory and have **.fsl** as extension. The **libload** function first tries to find an existing fasl file for loading.

fasl -> *<symbol>* **feature**

fasl feature is present when the fasl functions have been loaded in the System.

(fasl-dump *file-in file-out***)** -> *<object>* **function**

Convert Lisp file *file-in* in the corresponding fasl format file named *file-out*.

(fasl-table [*symb-table*]**)** -> *<object>* **function**

Internal function that sets or returns the current table used to read the current FALS-file. You should not call this function directly.

49 Trace Library.

The utilities described in this section are sufficiently complex and sufficiently dependent on the host environment that their complete definition is beyond the scope of this book. However, they are also sufficiently useful to warrant mention here. It is expected that every implementation will provide some version of these utilities, however clever or however simple.

trace **feature**

trace feature is present when the profile functions have been loaded in the System.

(trace *fn****)** -> *<object>* **function**

Invoking **trace** with one or more function-names (symbols) causes the functions named to be traced. Henceforth, whenever such a function is invoked, information about the call, the arguments passed, and the eventually returned values, if any, will be printed to the stream that is the value of **(standard-output)**.

(untrace *fn****)** -> <*object*> **function**

Invoking **untrace** with one or more function names will cause those functions not to be traced any more. Calling **untrace** with no argument forms will cause all currently traced functions to be no longer traced.

50 Internal Debugger.

OpenLisp has a simplified debugger that can be used to find the bugs of a program. Although it does not have all the appreciated features of a modern graphic debugger, it can be useful to keep tracks of simple bugs. Since the memory is more than limited on pure 16 bits MS-DOS, the debugger is not available for this environment.

debug -> <*symbol*> **feature**

debug is a feature indicating that the library has been loaded by the system. You can force this feature to be present by calling the function **debug**.

(debug *flag***)** -> <*symbol*> **function**

(debug t) makes the internal debugger active. **(debug nil)** removes activation of the debugger on errors. Under the control of the debugger toplevel loop, you can use the following keyword commands:

```
?    print this help.
:e   print environment.
:b   print current block.
:c   continue (if error is continuable).
:d   print stack depth.
:h   print history of calls.
:m   print error message.
:q   quit the debugger.
:s   print stack dump.
:t   print top stack history.
:u   up one block.
:v   print bindings.
:x   exit from OpenLisp.
```

Many commands may be run at the same time. That way, **:ubv** goes one block up; display this block and accessible variable form the current lexical point.

(stack-trace [*max-depth***])** -> <*list*> **function**

Returns a list that contains the current function calls. The option *max-depth* limits the number of calls that this function returns. You can call **stack-trace** in an error handler to display informations of the calling sequence up to the function where error occurs.

51 Performance Analysis.

The profiler is an analysis tool that you can use to examine the run-time behavior of your programs. By using profiler information, you can determine which sections of your code are working efficiently. The profiler can produce informations showing areas of code that are not being executed or that are taking a long time to execute. Because profiling is a tuning process, you should use the profiler to make your programs run better, not to find bugs. Once your program is fairly stable, you should start profiling to see where your code could perform better. Use the profiler to determine whether, an algorithm is effective (timing), a function is being called too many or too few times with respect to the problem domain (counting), or a piece of code is being covered by software testing procedures (coverage).

The profiler can be run from within the development environment.

profile -> *<symbol>* **feature**

profile feature is present when the profile functions have been loaded in the System.

(profile *fn***)** -> *<object>* **function**

Invoking **profile** with one function-name (symbol) causes the function named *fn* to be profiled. Henceforth, whenever such a function is invoked, information about the time of the call will be stored in an internal buffer.

(unprofile *fn***)** -> *<object>* **function**

Invoking **unprofile** with one or more function names will cause those functions not to be profiled any more. Calling **unprofile** with no argument forms will cause all currently profiled functions to be no longer traced.

(profile-all) -> *<object>* **function**

Invoking **profile-all** causes all interpreted functions to be profiled. Henceforth, whenever such a function is invoked, information about the time of the call will be stored in an internal buffer.

(unprofile-all) -> *<object>* **function**

Invoking **unprofile-all** removes profiling for all functions.

(profile-log *log-type***)** -> *<object>* **function**

Sort and display time and use informations about profiled functions denpening of *log-type* value :

time	sort by execution time.
call	sort by call number.
notused	only display unused functions.

52 Pretty Printer.

Pretty print library is used to re-format complex Lisp expressions in a more readable form.

print-pretty -> *<boolean>* **dynamic variable**

Controls whether the **OpenLisp** printer calls the pretty printer.

If it is **false**, the pretty printer is not used and a minimum of whitespace is output when printing an expression.

If it is **true**, the pretty printer is used, and the **OpenLisp** printer will endeavor to insert extra whitespace where appropriate to make expressions more readable.

print-pretty has an effect even when the value of ***print-escape*** is **false**.

pretty -> *<symbol>* **feature**

pretty feature is present when the pretty-printer functions have been loaded in the System.

(pretty *fn* **[***stream***])** -> *<object>* **function**
^Vfn -> *<object>* **macro character**

Format function named *fn* on the stream *stream*. If the second argument *stream* is not given, output goes to **(standard-output)**.

(pprint *form* **[***stream***])** -> <*object*> **function**

Format any Lisp expression *form* on the stream *stream*. If the second argument *stream* is not given, output goes to **(standard-output)**.

53 GC Functions.

The utilities described in this section are sufficiently complex and sufficiently dependent on the host environment that their complete definition is beyond the scope of this book. However, they are also sufficiently useful to warrant mention here. It is expected that every implementation will provide some version of these utilities, however clever or however simple.

---

gc-stats -> <*symbol*> **feature**

---

gc-stats feature is present when the gc functions have been loaded in the System.

(room [*flag***])** -> <*object*> **function**

---

room prints (using values returned by **gc** and **gcinfo**), to the stream **(standard-output)**, information about the state of internal storage and its management. This might include descriptions of the amount of memory in use and the degree of memory compaction, possibly broken down by internal data type if that is appropriate. The nature and format of the printed information is implementation-dependent. The intent is to provide information that may help a user to tune a program to a particular implementation.
(room nil) prints out a minimal amount of information. **(room t)** prints out a maximal amount of information. Simply **(room)** prints out an intermediate amount of information that is likely to be useful.

Example :

```
? (room t)
System name: openlisp, pointer size: 4
Type    Call     Init  MemInit     Used  MemUsed     Free  MemFree Free%
user       1        0        0        0        0        0        0  100%
cons       0    39280   314240     1607    12856    37673   301384   95%
symbol     0     3904   124928      549    17568     3355   107360   85%
string     0     3928    31424      580     4640     3348    26784   85%
vector     0     3928    31424      260     2080     3668    29344   93%
float      0        0        0        0        0        0        0  100%
integer    0        0        0        0        0        0        0  100%
heap       0   122880   122880    10836    10836   112044   112044   91%
Total      1        0   624896        0    47980        0   576916   92%
;; elapsed time = 0.22s, (1 gc).
= t
```

54 External Functions.

OpenLisp provides a simple way to write your own module without to worry about internal representation. With simple declaration, a C wrapper file will be generated for you (see **external.lsp** library). Calling **external-module** lisp function will generate this stub. The current version only supports the following C types : **int**, **double**, **char**, **char ***, **void *** and **bool** which are mapped to **<integer>**, **<float>**, **<character>**, **<string>**, **<external>**, **<boolean>** respectively. The generated stub file will automatically convert arguments in both directions.

---

external -> <*symbol*> **feature**

---

external feature is present when the external definition functions have been loaded in the System.

(external-module *module stub headers* **[***decl****])** -> <*object*> **function**

This function creates two files that are used to extend OpenLisp with an external module named *module* written in C (or C++). The first generated file named *stub*`.h` contends declaration for the sub functions as well as a function named **ol***module***init()** that is used to initialize this module. The second file, named *stub*`.c` has the definition for the stub functions that will be called by Lisp. *decl* is a list of optional filenames to be inclued by *stub*`.c` . When *headers* is non-**nil**, it contains a list of header files to be inclued. Each declared function *decl* has one the following form:

> **(***retval external-name* **(***arg-type*$_1$ ... *arg-type*$_n$**))**
> **(***retval* **(***external-name lisp-name***)** **(***arg-type*$_1$ ... *arg-type*$_n$**))**
> **(***retval* **(***accessor lisp-name***)** *var***)**

The first line declares an external function *external-name* and returning a value of list type *retval*. This function accepts *n* Lisp parameter *arg-type*$_i$. In that case, Lisp will use the same name as in C. The second line, declares the same function but given a different Lisp name *lisp-name*. The third line declares an accessor function (**:reader :writer**) to a global C variable.

Only the following types are yet supported for *retval* and *arg-type*$_i$:

Lisp type	C type	Meaning
`<integer>`	`int` or `long`	native integer type
`<float>`	`double`	floating point number
`<string>`	`char*`	null terminated C string
`<character>`	`char`	8 bits character
`<external>`	`void*`	external pointers
`<null>`	`void`	only used as a return type (always **nil**)
`<boolean>`	`int`	map 0 to **nil** and **t** to all other values
`<object>`	`void*`	uninterpreted pointer

You can also declare global pointer, float and integer constants that match C object that are globally defined (possibly with a `#define`) using the **:constant** declaration:

> **(:constant** <*type*> **(***lisp-symbol-name c-name***)** *[***:if-defined***]***)**
>
> or
>
> **(:constant** <*type*> *c-name* *[***:if-defined***]***)**

With the latest form, the Lisp symbol name is the same as in C.
If **:if-defined** is supplied as 4th argulent, definition may not exist and is surrounded by **#if defined(***c-name***)** .. **#endif**.

Example :

```
(external-module foo stubfoo (<bar.h> "gee.h")
      (:constant  <integer>  (|*MAX-FOO*| |MAX_FOO|)
      (:constant  <float>    |MAX_BAR| :if-defined)
      (:constant  <external> (*stdout* stdout))
      (<integer>  (:reader get-current-value) |current|)
      (<integer>  |Foo| (<integer> <float> <string>))
      (<external> |bar| ())
      (<object>   (|yab| yab-function) ())
      (<integer>  |gee| (<integer> <integer> <integer> <integer>)))
```

You can include some C code in a special `:code` section:

```
(external-module foo stubfoo (<bar.h> "gee.h")
      (:code
         "static void"
         "myfun( … )"
         "{"
         "}"
      )
```

55 Multiple Values

Normally, multiple values are not used by **OpenLisp** itself. This package is provided to help tansition from other Lisp dialects.
Special forms are required both to *produce* multiple values and to *receive* them. If the caller of a function does not request multiple values, but the called function produces multiple values, then the first value is given to the caller and all others are discarded; if the called function produces zero values, then the caller gets **nil** as a value.
The primary primitive for producing multiple values is **values**, which takes any number of arguments and returns that many values. If the last form in the body of a function is a **values** with three arguments, then a call to that function will return three values. No built-in can return multiple values.
The special forms and macros for receiving multiple values are as follows

---

multiple-values -> *<symbols>* **feature**

---

multiple-values feature is present when the multiple values functions have been loaded in the System.

---

multiple-value-limit -> *<integer>* **constant**

The value of **multiple-values-limit** is a positive integer that is the upper exclusive bound on the number of values that may be returned from a function. This bound depends on the implementation but will not be smaller than 20.

(values *arg_1 .. arg_N***)** -> *<object>* **function**

All of the arguments are returned, in order, as values.

Example:

```
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x)))

(multiple-value-bind (r theta) (polar 3.0 4.0)
  (vector r theta))
   => #(5.0 0.9272952)
```

The expression **(values)** returns zero values. This is the standard idiom for returning no values from a function.
Sometimes it is desirable to indicate explicitly that a function will return exactly one value. For example, the function

```
(defun foo (x y)
  (values (+ x y) y))
```

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

```
(defun foo (x y)
```

```
(values (bar (+ x y) y)))
```

This works because **values** returns exactly *one* value for each of its argument forms; as for any function call, if any argument form to **values** produces more than one value, all but the first are discarded.
There is absolutely no way for a caller to distinguish between returning a single value in the ordinary manner and returning exactly one ``multiple value." For example, the values returned by the expressions **(+ 1 2)** and **(values (+ 1 2))** are identical in every respect: the single value 3.

(values-list *list***)** -> <*object*> **function**

All of the elements of *list* are returned as multiple values. For example:
(values-list (list a b c)) == (values a b c)
In general,
(values-list list) == (apply #'values list)
but **values-list** may be clearer or more efficient.

(multiple-value-list *form***)** -> <*object*> **macro**

---

multiple-value-list evaluates *form* and returns a list of the multiple values it returned.

Example:

```
(multiple-value-list (values -3 4))   => (-3 4)
```

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

(multiple-value-call *fun body****)** -> <*object*> **macro**

---

multiple-value-call first evaluates *function* to obtain a function and then evaluates all of the *form*s. All the values of the *form*s are gathered together (not just one value from each) and are all given as arguments to the function. The result of **multiple-value-call** is whatever is returned by the function.

Example:

```
(+ (values 5 3) (values 1 4))         => 6 ;; (+ 5 1)
(multiple-value-call #'+ (values 5 3) (values 1 4))
                                      => 12 ;; (+ 5 3 1 4)

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

(multiple-value-prog1 *fisrt body****)** -> <*object*> **macro**

---

multiple-value-prog1 evaluates the first *form* and saves all the values produced by that form. It then evaluates the other *form*s from left to right, discarding their values. The values produced by the first *form* are returned by **multiple-value-prog1**. See **prog1**, which always returns a single value.

(multiple-value-bind *variables form body****)** -> <*object*> **macro**

The *values-form* is evaluated, and each of the variables *var* is bound to the respective value returned by that form. If there are more variables than values returned, extra values of **nil** are given to the remaining variables. If there are more values than variables, the excess values are simply discarded. The variables are bound to the values over the execution of the forms, which make up an implicit **progn**.

Example:

```
(multiple-value-bind (x) (values 5 3) (list x))           => (5)
(multiple-value-bind (x y) (values 5 3) (list x y))       => (5 3)
(multiple-value-bind (x y z) (values 5 3) (list x y z))  => (5 3 nil)
```

(multiple-value-setq *variables form***)** -> <*object*> **macro**

The *variables* must be a list of variables. The *form* is evaluated, and the variables are *set* (not bound) to the values returned by that form. If there are more variables than values returned, extra values of **nil** are assigned to the remaining variables. If there are more values than variables, the excess values are simply discarded. **multiple-value-setq** always returns a single value, which is the first value returned by *form*, or **nil** if *form* produces zero values.

(nth-value *n form***)** -> <*object*> **macro**

The argument forms *n* and *form* are both evaluated. The value of *n* must be a non-negative integer, and the *form* may produce any number of values. The integer *n* is used as a zero-based index into the list of values. Value *n* of the *form* is returned as the single value of the **nth-value** form; **nil** is returned if the *form* produces no more than *n* values.

56 Virtual Terminal.

On some systems (MS-DOS, NT, 9x and Windows), **OpenLisp** has been extended to support a kind of Virtual Character Mode Terminal (**virtty**). It can be used to build very simple interface such as editor (see **lib/mine.lsp**) or simple game (see **contrib/hanoi.lsp**). This module, not part of "standard **OpenLisp**", should not be used to write real interfaces. For that purpose, graphic C/C++ generators should be considered. That way, **OpenLisp** has been successfully integrated with Windows API, Microsoft MFC and Ilog Views class library. Functionalities of this module are taken after the **virtty** package form Le-Lisp.

virtty -> <*symbol*> **feature**

virtty feature is present when the virtual terminal functions have been loaded in the System.

(typrologue) -> <*object*> **function**

Set the terminal in some "special mode" suitable for drawing characters.

(tyepilogue) -> <*object*> **function**

Reset the terminal to the "standard mode".

(tyxmax) -> <*object*> **function**

Returns the maximum number of character that can fit on a single line starting from 0 (*i.e.* a 80 characters width will return 79).

(tyymax) -> <*object*> **function**

Returns the maximum number of character that can fit on a single column starting from 0 (*i.e.* a 25 characters height will return 24).

(tycls) -> <*object*> **function**

Clears the entire screen.

(tycleol) -> <*object*> **function**

Clears the current line form point to end of line.

(tyflush) -> *<object>* **function**

Flush unsent characters.

(tybeep) -> *<object>* **function**

Sounds the beeper.

(tyi) -> *<object>* **function**

Reads a single character without echo.

(tys) -> *<object>* **function**

Tests whether or not a character can be read form keyboard.

(tycn *char***)** -> *<object>* **function**

Draws character *char* at current position.

(tycursor *x y***)** -> *<object>* **function**

Moves the cursort to *x*, *y* location on screen.

(tyshowcursor *flag***)** -> *<object>* **function**

Hides, if *flag* is false or shoes if *flag* is true the hardware cursor.

(tyo *o***)** -> *<object>* **function**

Draws the object *o* (a character, a string or a list of characters) at current position.

(tyco *x y o***)** -> *<object>* **function**

Draws the object *o* (a character, a string or a list of characters) at position *x, y*.

(tystring *string n***)** -> *<object>* **function**

Draws the first *n* characters of string *string* at current position.

(tyattrib *n***)** -> *<object>* **function**

Set, if flag is **t**, or reset if flag is **nil** the drawing attribute (it can be another color, a blinking mode, …).

57 OpenLisp Dynamic Server Page

OpenLisp has a special mode called ODSP "OpenLisp Dynamic Server Page" that allows you to generate dynamic HTML/XML page with embedded lisp code. And ODSP page is a standard HTML/XML text page where code between **`<?openlisp`** and **`?>`** tags is handled directly by **OpenLisp**. When **OpenLisp** is in ODSP mode, it echoes to console the HTML/XML page up to **`<?openlisp`** tag. The code is then evaluated until and end tag **`?>`** is encountered. Then, echo of standard HTML/XML continues until the next **`<?openlisp`** tag or the end of file.

You can start ODSP mode with **`–odsp`** flag on the command line like:

```
$ openlisp –odsp file.odsp
```

or, on systems with unix shell, by directly calling **`file.odsp`** if you add line like:

```
#!/usr/bin/env openlisp –odsp
```

on top of your HTML/XML file.

Example:

```
#!/usr/bin/env openlisp -odsp
<?xml version="1.0" encoding="UTF-8"?>
<!-- OpenLisp Dynamic Server Page Sample - (c) C. Jullien 2001/09/12 -->

<!DOCTYPE html
          PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
        "DTD/xhtml1-frameset.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
 <title>OpenLisp Dynamic Server Page Sample</title>
 <meta http-equiv="Content-Type" content="text/html" />
 <meta http-equiv="Pragma"       content="no-cache" />
 <meta name="Generator"          content="OpenLisp Dynamic Server Page" />
 <meta name="robots"             content="noindex,follow" />
```

```
</head>

<body>
 <p>
  Openlisp v<?openlisp (prin (version)) ?> Dynamic Server Page,
  computes Fib serie:
  <table border="1"
         bordercolor="#000000"
       cellpadding="4"
       cellspacing="1">
   <tbody>
    <?openlisp

     ;; nice way to WEB around!!!

     (require 'cgi) ;; not really needed for this sample

     (defun fib (n)
      ;; Standard function with integer argument
      (cond ((= n 1) 1)
            ((= n 2) 1)
            (t (+ (fib (1- n)) (fib (- n 2))))))

     (for ((i 10 (1+ i)))
        ((> i 20))
        (format (standard-output)
              "<tr><td>(fib ~D)</td><td>~D</td></tr>~%" i (fib i)))

    ?>
   </tbody>
  </table>
 </p>
</body>

</html>
```

58 Install OpenLisp as a NPH-CGI/1.1 compliant on-the-fly filter.

This is the standard method of using **OpenLisp** as a SSSL (**`Server-Side-Scripting-Language`**). Here the original OpenLisp interpreter gets used by just installing the stand-alone program "openlisp" as a on-the-fly filter for Apache. This shell is NPH (No-Parse-Headers) CGI module. It means that it must process headers itself.

First, you must install the stand-alone program **OpenLisp**.

In short:

```
$ ./configure --prefix=/path/to/openlisp/
$ make
$ make install
```

Create the following **`nph-openlisp`** (you *MUST!* start this name with **`nph-`**) script to the CGI directory of Apache and use the NPH-CGI/1.1 compliant interface:

```
#!/bin/sh
#ident "@(#)nph-openlisp (c) C. Jullien 2001/09/18"

# OpenLisp NPH-CGI/1.1 Apache compliant on-the-fly filter.
```

```
echo HTTP/1.0 200 Script results follow
echo Server: OpenLisp Dynamic Server Page
echo Content-Type: text/html

exec /usr/bin/env openlisp -odsp $PATH_TRANSLATED
```

Change permissions:

```
$ chown root /path/to/apache/cgi-bin/nph-openlisp
$ chmod u+s  /path/to/apache/cgi-bin/nph-openlisp
```

Finally configure Apache to process all pages with extension '`.odsp`' via the **OpenLisp** NPH-CGI/1.1 program. To accomplish this you have to add the following to your `httpd.conf` file of Apache (on Red Hat distribution, this file is located at `/etc/httpd/conf/httpd.conf`):

```
AddType  application/x-httpd-openlisp .odsp .osp .lsp .lap
Action   application/x-httpd-openlisp /cgi-bin/nph-openlisp
```

You must restart Apache by calling `apachectl restart` or, on Red Hat, `service http restart`.

Now test it by copying the file from the `tst/test.odsp` of the **OpenLisp** distribution to an area that is accessible by Apache and request the file `test.odsp` through your WEB browser. It should be implicitly piped *on-the-fly* through the **OpenLisp** system.

59 OpenLisp and mod_lisp

60 What is mod_lisp?

mod_lisp is an Apache (http://httpd.apache.org/) module to write dynamic web servers and applications. The source (FreeBSD style license), lisp examples and pre-compiled binaries for FreeBSD, Linux and Win32 are on the mod_lisp web site (http://www.fractalconcept.com/asp/mod_lisp).

61 Preparing Apache with mod_lisp

You must make the **mod_lisp2.c** source code and, from **mod_lisp OpenLisp** directory, run “make” or execute the following command:

```
$ apxs -i -c mod_lisp2.c
```

The apxs command comes from Apache development package. If you use a Red Hat Linux distribution, you can use:

```
$ yum install httpd-devel
```

to install this package.

After compiling, follow the procedure in the **mod_lisp** documentation to properly edit **httpd.conf** to load the library and call the lisp handler. In the case of this site, the handler location is lispy:

```
LoadModule   mod_lisp    modules/mod_lisp2.so
  ...
  ...
AddModule mod_lisp.c
  ...
  ...
LispServer IP.IP.IP.IP  3000 "lisp"
  ...
```

```
    ...
  < Location /lisp >
      SetHandler   lisp-handler
  </Location >
```

Restart apache (`apachectl restart`), launch **OpenLisp** with `modlisp-openlisp.lsp` and try the service.

```
;; OpenLisp v8.4.0 (Build: 3874) by C. Jullien [Sep 24 2006 - 20:01:59]
;; Copyright (c) 1988-2006.
;; System 'linux' (32-bit) on 'fc5.eligis.com', ASCII.
;; Reading startup ..
;; God thank you, OpenLisp is back again!
? (load "modlisp-openlisp.lsp")
;; elapsed time =  0.059s, (0 gc).
= modlisp-openlisp.lsp
? (modlisp-server)
```

Any url using “**`lisp`**” like “http://127.0.0.1/**lisp**/foo” will be redirected to your lisp process.

62 GNU-Emacs integration

63 What is Emacs.

Emacs is a family of editors that have a long common story with Lisp. As such, Emacs is the preferred Lisp source code editor for most Lisp developers; however, it is not easy for an Emacs neophyte to get an Emacs environment set up and configured properly for Lisp development. The GNU Emacs editor (http://www.gnu.org/software/emacs/) has been customized to provide an efficient IDE environment for **OpenLisp**.

64 Install Emacs.

You can find a GNU Emacs precompiled distribution for most operating systems including Linux and Windows. For example, on Windows you can download EmacsW32 and binaries on http://ourcomments.org/Emacs/EmacsW32.html. Edit or create an .emacs initialization file and add the following lines at the bottom:

```
(custom-set-variables
  ;; custom-set-variables was added by Custom.
  ;; If you edit it by hand, you could mess it up, so be careful.
  ;; Your init file should contain only one such instance.
  ;; If there is more than one, they won't work right.
 '(safe-local-variable-values
        (quote ((Syntax  . EmacsLisp)
                (Mode    . LISP)
                (Package . LISP)
                (Base    . 10)
                (Syntax  . ISLISP)))))

(defun check-openlisp (path)
   ;; check if openlisp.el exists and load it.
   (let ((path      (getenv path))
         (found    nil)
```

```
         (openlisp nil))
        (when path
              (setq openlisp (concat path "/emacs/openlisp.el"))
              (when (file-exists-p openlisp)
                    (load openlisp)
                    (setf found t)))
        found))

(or
    (check-openlisp "HOME")
    (check-openlisp "HOMEPATH")
    (check-openlisp "OPENLISP"))
```

65 Use OpenLisp inside Emacs.

Emacs is mainly used as a text editor to create or modify your lisp files but you can also use Emacs to execute and debug your programs. The **OpenLisp** REPL (Read Eval Print Loop) can be run inside an Emacs windows and with appropriate key bindings, you can interact between your source code and **OpenLisp** topelevel. To run **OpenLisp** inside Emacs, Execute “M-x run-lisp” where M-x is the sequence Escape followed by small ‘x’ to get a new Emacs Windows with fully functional **OpenLisp** system. It’s generally convenient to have two Windows, one for edition the other one to run your code inside Lisp.

66 Special commands for OpenLisp mode

When you edit your Lisp code, Emacs has the specific key binding to interact with **OpenLisp**:

C-xC-e	Evaluate Lisp expression
C-xC-h	Evaluate buffer
C-xC-m	Macroexpand expression
C-xC-z	Go to OpenLisp window
C-cC-e	Evalute defun form
C-cC-r	Evaluate region
C-cC-c	Compile defun
C-cC-z	Go to OpenLisp window
C-cC-l	Load file
C-cC-k	Compile file

67 LAP Format

The compiler uses an internal LAP (Lisp Assembly Processor) format for the compiled code. At load time, this format is converted in a form suitable for execution. This chapter describes the format of this virtual assembler.
Internally, the LAP machine uses 4 public registers named A1, A2, A3, A4 and an internal register named A0 which cannot be used by any instruction. By default, instructions that produce a result store the value in A1 register.

(ADD *reg reg/imm***)** **LAP function**

Add in A1 the contents of first register (an integer) with the value of *reg/imm.*

(ADD1 *reg***)** **LAP function**

Increment in A1 the contents of register (an integer value) by one.

(ADJSTK *n***)** **LAP function**

Adjust the stack by the value of the positive integer *n.* A1 is not modified by this instruction.

(ASSOC *reg reg/imm***)** **LAP function**

Returns in A1 the **assoc** pair of reg in A-List in *reg/imm.*

(BLOCK *label***)** **LAP function**

Setup a new BLOCK block with tag named *label.* A1 is not modified by this instruction.

(CAAR *reg***)** **LAP function**

Returns in A1 the caar of register *reg.*

(CAAR *reg***)** **LAP function**

Returns in A1 the cadr of register *reg.*

(CALL *n***)** **LAP function**

Calls a pushed function with *n* – 1 arguments pushed from left to right. The result is stored in A1. After this call, the stack is cleared by the number of pushed values.

(CALL-REG *fname* [[*reg1*] *reg2*]**)** **LAP function**

Calls a function *fname* with optional register arguments in *reg1* and *reg2*. Returns result in A1.

(CALL-SUBR *fname* [[*reg1*] *reg2*]**)** **LAP function**

Calls a subrX function *fname* with optional register arguments in *reg1* and *reg2*. Returns result in A1.

(CAR *reg***)** **LAP function**

Returns in A1 the car of register *reg.*

(CATCH *label***)** **LAP function**

Setup a new CATCH block with tag named *label.* A1 is not modified by this instruction.

(CDAR *reg***)** **LAP function**

Returns in A1 the cdar of register *reg.*

(CDDR *reg***)** **LAP function**

Returns in A1 the cddr of register *reg.*

(CDR *reg***)** **LAP function**

Returns in A1 the cdr of register *reg.*

(CHECKTYPE *n***)** **LAP function**

Check if register A1 is of type *n.* If A1 satisfies the type *n*, upon return A1 has the value **`t`** and **`nil`** otherwise.

0	symbolp
1	consp
2	numberp
3	integerp
4	floatp
5	atom
6	functionp
7	characterp
8	stringp
9	classp
10	vectorp
11	arrayp
12	bignump
13	variablep
14	streamp
15	boundp
16	socketp

(CLOSURE *symb***)** **LAP function**

Returns in A1 a closure for the function named *symb.*

(CONS *reg reg/imm***)** **LAP function**

Cons in A1 the contents of first register with the value of *reg/imm.*

(CONVERT *n***)** **LAP function**

Convert register A1 to type numbered *n.* Returns result in A1.

0	<character>
1	<float>
2	<general-vector>
3	<integer>
4	<list>
5	<string>
6	<symbol>
7	<external>
8	<rational>
9	<simple-bit-vector>

(DECR-REF *n1 n2***)** **LAP function**

Decrement the value of local variable at offset *n2* from the block *n1.* A1 contains the modified value.

(DECR-SREF *offset***)** **LAP function**

Decrement the stack value at offset (positive value means function parameter, negative value means local variable).

(DIV *reg reg/imm***)** **LAP function**

Divide in A1 the contents of first register with the value of *reg/imm.*

(DIVN *reg reg/imm***)** **LAP function**

Integer divide in A1 the contents of first register with the value of *reg/imm.*

(DREF *reg symb***)** **LAP function**

Loads in register *reg* the dynamic value of symbol *symb*.

(DSET *symb***)** **LAP function**

Sets the dynamic value of symbol *symb* with the value of register A1.

(DYNAMIC-LET *symb***)** **LAP function**

Setup a new dynamic block of *n* variables pushed on stack. A1 is not modified by this instruction.

(ELT *reg reg/imm***)** **LAP function**

Loads in A1 the element of vector in *reg* at position *reg/imm*.

(END) **LAP function**

This pseudo-instruction terminates the LAP instruction list. It must be the last instruction of a function.

(ENTER *n1 n2***)** **LAP function**

Enter a new function block with *n1* parameters and *n2* local variables. A1 is not modified by this instruction.

(EQ *reg reg/imm***)** **LAP function**

Compares in A1 if *reg* and *reg/imm* are the same object.

(EQUAL *reg reg/imm***)** **LAP function**

Compares in A1 if *reg* and *reg/imm* are **equal**.

(EQN *reg reg/imm***)** **LAP function**

Returns in A1 if *reg* and *reg/imm* are the same integer.

(EXIT-BLOCK) **LAP function**

Exit from a BLOCK block. A1 is not modified by this instruction.

(EXIT-CATCH) **LAP function**

Exit from a CATCH block. A1 is not modified by this instruction.

(EXIT-DYNAMIC *n***)** **LAP function**

Exit from a DYNAMIC block with *n* argumenst. A1 is not modified by this instruction.

(EXIT-HANDLER) **LAP function**

Exit from a HANDLER block. A1 is not modified by this instruction.

(EXIT-PROTECT) **LAP function**

Exit from a PROTECT block. A1 is not modified by this instruction.

(FENTRY *n1 n2 n3***)** **LAP function**

Declares and entry function with *n1* parameters, *n2* is non-0 if the function has been declared with &rest. The last parameter *n3* is the number of local parameters on the stack.

(FUNCTION-VALUE *symb***)** **LAP function**

Returns the functional object associated with the symbol *symb.*

(GADD *reg reg/imm***)** **LAP function**

Add in A1 the contents of first register (any number type) with the value of *reg/imm.*

(GADD1 *reg***)** **LAP function**

Increment in A1 the contents of first register (any number type).

(GDIV *reg reg/imm***)** **LAP function**

Divide in A1 the contents of first register (any number type) with the value of *reg/imm.*

(GDECR-REF *n1 n2***)** **LAP function**

Decrement the value of local variable at offset *n2* from the block *n1.* A1 contains the modified value.

(GDECR-SREF *offset***)** **LAP function**

Decrement the stack value at offset (positive value means function parameter, negative value means local variable).

(GE *n***)** **LAP function**

Compares in A1 if the *n* integer arguments pushed on stacks are greater or equal pair wise.

(GEQN *reg req/imm***)** **LAP function**

Returns in A1 if *reg* and *reg/imm* are the same number.

(GGE *n***)** **LAP function**

Compares in A1 if the *n* numeric arguments pushed on stacks are greater or equal pair wise.

(GGT *n***)** **LAP function**

Compares in A1 if the *n* numeric arguments pushed on stacks are greater pair wise.

(GINCR-REF *n1 n2***)** **LAP function**

Increment the value of local variable at offset *n2* from the block *n1.* A1 contains the modified value.

(GINCR-SREF *offset***)** **LAP function**

Increment the stack value at offset (positive value means function parameter, negative value means local variable).

(GLE *n***)** **LAP function**

Compares if the *n* numeric arguments pushed on stacks are less than or equal pair wise.

(GLT *n***)** **LAP function**

Compares if the *n* numeric arguments pushed on stacks are less than pair wise.

(GMUL *reg reg/imm***)** **LAP function**

Multiply in A1 the contents of first register (any number type) with the value of *reg/imm.*

(GNEQN *reg req/imm***)** **LAP function**

Returns in A1 if *reg* and *reg/imm* are the two distinct number.

(GREF *reg symb***)** **LAP function**

Returns in register *reg* the global value of symbol *symb.*

(GSET *symb***)** **LAP function**

Sets the global value of symbol *symb* with A1.

(GSUB *reg reg/imm***)** **LAP function**

Substract in A1 the contents of first register (any number type) with the value of *reg/imm.*

(GSUB1 *reg***)** **LAP function**

Decrement in A1 the contents of first register (any number type).

(GT *n***)** **LAP function**

Compares in A1 if the *n* integer arguments pushed on stacks are greater pair wise.

(INCR-REF *n1 n2***)** **LAP function**

Increment the integer value of local variable at offset *n2* from the block *n1.* A1 contains the modified value.

(INCR-SREF *offset***)** **LAP function**

Increment the integer stack value at offset (positive value means function parameter, negative value means local variable).

(JEQ *label cst***)** **LAP function**

Jumps to label *label* if A1 is equal to constant *cst*.

(JNEQ *label cst***)** **LAP function**

Jumps to label *label* if A1 is not equal to constant *cst*.

(JNIL *symb***)** **LAP function**

Jumps to label *symb* if A1 is null.

(JTRUE *symb***)** **LAP function**

Jumps to label *symb* if A1 is not null.

(JUMP *symb***)** **LAP function**

Jumps to label *symb*.

(LE *n***)** **LAP function**

Compares if the *n* arguments pushed on stacks are less or equal pair wise.

(EQ *reg reg/imm***)** **LAP function**

Compares in A1 if *reg* and *reg/imm* are the same object.

(LEAVE *n***)** **LAP function**

Leave from a function with *n* local arguments. A1 is not modified by this instruction.

(LENGTH *reg***)** **LAP function**

Returns in A1 the length of object pointed by *reg.*

(LENTRY *n1 n2 n3***)** **LAP function**

Declares a local function with *n1* actual parameters, *n2* is non-0 if the function has been declared with &rest. The last parameter *n3* is the number of local parameters on the stack.

(LIST *n***)** **LAP function**

Creates in A1 a list with the *n* arguments pushed on stack. The stack is cleared by this instruction.

(LOCAL-CALL *n***)** **LAP function**

Calls a local function (as created by labels or flet) with *n* arguments pushed on stack. The result is stored in A1. After this call, the stack is cleared by the number of pushed values.

(LOCAL *n***)** **LAP function**

Returns in A1 the value of *n*th local variable.

(LREF *n1 n2***)** **LAP function**

Loads in A1 the value of variable at offset *n2* in display *n1*.

(LSET *n1 n2***)** **LAP function**

Sets with the value in A1 the variable at offset *n2* in display *n1*.

(LT *n***)** **LAP function**

Compares if the *n* arguments pushed on stacks are less than pair wise.

(MENTRY *n1 n2 n3***)** **LAP function**

Declares a macro function with *n1* actual parameters, *n2* is non-0 if the function has been declared with &rest. The last parameter *n3* is the number of local parameters on the stack.

(MOVE *reg reg/imm***)** **LAP function**

Copy the into the first register the value of *reg/imm.*

(MUL *reg reg/imm***)** **LAP function**

Copy the into A1 the value of the first register the value of *reg/imm.*

(NEQ *reg reg/imm***)** **LAP function**

Compares in A1 if *reg* and *reg/imm* are not the same object.

(NEQN *reg reg/imm***)** **LAP function**

Copy the into the first register the value of *reg/imm.*

(NEXT-REF *n1 n2***)** **LAP function**

Makes the value of local variable at offset *n2* from the block *n1* point to its CDR. A1 contains the modified value.

(NEXT-SREF *offset***)** **LAP function**

Advance in the list, the stack value at offset (positive value means function parameter, negative value means local variable).

(NOP) **LAP function**

No operation.

(NULL *reg***)** **LAP function**

Returns t in A1 if the value of register *reg* is null and nil otherwise.

(PARAM *n***)** **LAP function**

Returns in A1 the value of *n*th local variable.

(POP *reg***)** **LAP function**

Remove into the register *reg* the top of the stack.

(PROTECT *reg***)** **LAP function**

Setup a new UNWIND-PROTECT block with the cleanup function set in register *reg.*

(PUSH *reg/imm***)** **LAP function**

Push on the stack the value of *reg/imm.* A1 is not modified by this instruction.

(PUSH-FUNCTION *symb***)** **LAP function**

Push on the stack the functional value of symbol *sym.* A1 is not modified by this instruction.

(PUSH-LOCAL *symb***)** **LAP function**

Push on the stack the functional value of a locally defined function *sym.* A1 is not modified by this instruction.

(PUSH-REF *n1 n2***)** **LAP function**

Push on the stack the value of local variable at offset *n2* from the block *n1.* A1 is not modified by this instruction.

(PUSH-DREF *symbol***)** **LAP function**

Push on the stack the value of dynamic variable named *symbol.* A1 is not modified by this instruction.

(PUSH-GREF *symbol***)** **LAP function**

Push on the stack the value of global variable named *symbol.* A1 is not modified by this instruction.

(PUSH-SREF *offset***)** **LAP function**

Push the stack value at offset (positive value means function parameter, negative value means local variable).

(RECURSE *n***)** **LAP function**

Call recursively the current function with *n* argument pushed on the stack.

(RETURN-FROM *label***)** **LAP function**

Return from label *label*.

(RETURN) **LAP function**

Return form the current function. A1 is not modified by this instruction.

(SET-CAR *reg reg/imm***)** **LAP function**

Change the car of register *reg/imm* with value of *reg*.

(SET-CDR *reg reg/imm***)** **LAP function**

Change the cdr of register *reg/imm* with value of *reg*.

(SET-ELT *reg1/imm reg2 reg3/imm***)** **LAP function**

Set the value form *reg1/imm* into the sequence *reg2* indexed by *reg3/imm*.

(SET-LOCAL *n***)** **LAP function**

Set the value of *n*th local variable with the content of A1 register.

(SET-PARAM *n***)** **LAP function**

Set the value of *n*th function parameter with the content of A1 register.

(SUB *reg reg/imm***)** **LAP function**

Subtract in A1 the first register value with the value of *reg/imm*.

(SUB1 *reg***)** **LAP function**

Subtract 1 on A1 the register value of *reg*.

(TAILREC *n***)** **LAP function**

Call tail-recursively the current function with *n* argument pushed on the stack. This instruction does not return.

(THROW *reg reg/imm***)** **LAP function**

Throw value *reg/imm* form the CATCH block *reg*.

(TRACE *imm***)** **LAP function**

Display the immediate value *imm*.

(WITH-HANDLER *reg***)** **LAP function**

Setup a WITH-HANDLER block with the handler function in register *reg*.

68 C mapping of Lisp objects

This chapter explains some **OpenLisp** internals.

69 Internal representation

The type of a basic lisp object is determined directly by its address. **OpenLisp** uses a “low tag” scheme that codes the objet type in the 4th lowest bits of its address. Given a 32 bits processor you have the following representation.

31-28	27 - 24	23 - 20	19 - 16	15 - 12	11 – 8	7 - 4	3	2	1	0	
X	X	X	X	X	X	X	0	0	0	0	INTEGER
X	X	X	X	X	X	X	0	0	0	1	FLOAT
X	X	X	X	X	X	X	0	0	1	0	STRING
X	X	X	X	X	X	X	0	0	1	1	FLOAT
X	X	X	X	X	X	X	0	1	0	0	CONS
X	X	X	X	X	X	X	0	1	0	1	FLOAT
X	X	X	X	X	X	X	0	1	1	0	VECTOR
X	X	X	X	X	X	X	0	1	1	1	FLOAT
X	X	X	X	X	X	X	1	0	0	0	SYMBOL
X	X	X	X	X	X	X	1	0	0	1	FLOAT
X	X	X	X	X	X	X	1	0	1	0	STRING
X	X	X	X	X	X	X	1	0	1	1	FLOAT
X	X	X	X	X	X	X	1	1	0	0	CONS
X	X	X	X	X	X	X	1	1	0	1	FLOAT
X	X	X	X	X	X	X	1	1	1	0	VECTOR
X	X	X	X	X	X	X	1	1	1	1	FLOAT

That way, using a fast mask operation, we can know the exact type of an object:

FLOAT	☞	(obj & 0x00000001) != 0x01
INTEGER	☞	(obj & 0x0000000F) == 0x00
CONS	☞	(obj & 0x00000007) == 0x04
SYMBOL	☞	(obj & 0x0000000F) == 0x08
STRING	☞	(obj & 0x00000007) == 0x02
VECTOR	☞	(obj & 0x00000007) == 0x06

As you may notice, integers don't allocate space but are limited to range +/-2^{n-5}. There is also a "boxed" integer representation to store values out of this range. Generally, complex types like streams, hash tables, external pointers… use a basic vector object with the specific type in its extended type field. Concrete implementation details are beyond the scope of this document.

64bit processors, because of wider alignment, use five bits for tags but the general principle remains the same.

70 Objects creation

POINTER
olnewsymbol(CLCHAR *`name`);

Returns the symbol object in current package with print name *name*. If this symbol exists, it is simply returned. Otherwise a new object is created.
Example:

```
mysym = olnewsymbol( nil, LCSTR("mysymb") );
```

POINTER
olallocstring(POINTER `len`, POINTER `init`);

Returns a new lisp string of *len* characters all initialized to *init* (a character object).

POINTER
olallocvector(POINTER `len`, POINTER `init`);

Returns a new lisp vector of *len* objects all initialized to *init* (any lisp object).

POINTER
olcons(POINTER `car`, POINTER `cdr`);

Returns a new dotted-pair (`car . cdr`).

POINTER
olmakefixnum(int `i`);

Returns a new lisp integer from C integer i.

POINTER
olmakefloat(DOUBLE `f`);

Returns a new lisp float from C float *f*.

71 Calling Lisp code from C

POINTER
olevalbuffer(LCHAR *`lispexp`);

Call the internal evaluator with *lispexp* which should be a string that correspond to a valid Lisp expression.

LCHAR *
olstringevalbuffer(LCHAR *`lispexp`, LCHAR *`res`, int `size`);

Call the internal evaluator with *lispexp* which should be a string that correspond to a valid Lisp expression. The result is stored in the given *res* string buffer having a length of *size* characters. If *res* is **NULL**, *size* is ignored and the string is allocated using **olhookalloc** (malloc in general). Only in that case, the returned string must be free by the caller using **olhookfree** function.

Example1:

```
#define OLPRMAXBUF 1024

LCHAR buf[OLPRMAXBUF];
LCHAR *res;

res = olstringevalbuffer( LCSTR("(fib 20)"), buf, OLPRMAXBUF );
```

Example2:

```
LCHAR *res;

res = olstringevalbuffer( LCSTR("(fib 20)"), NULL, 0 );
...
olhookfree( res );
```

POINTER
ollispcall(POINTER *fun*, **POINTER** *arg1*, **...**, **POINTER** *argN* **);**

When you have direct access to the function and its arguments, **ollispcall** is much more efficient. It calls *fun* function using a mechanism similar to **funcall**.

Example:

```
fib20 = olevalbuffer( LCSTR("(fib 20)") );
```

or

```
fibfn = olmakesymbol( nil, LCSTR("fib") );
fib20 = ollispcall( fibfn, olmakefix( 20 ) );
```

72 Source file contents

OpenLisp is written in several C source files. Some of them (**`olinit`**, **`gc`**, **`eval`**, **`function`** ..) are part of the kernel and should ***never*** be modified, some other (**`openlisp`**, **`toplevel`**, **`debug`**, ..) are there to provide a usable "standard" Lisp environment. Last, the remaining files are only provided as "demonstrations" of **OpenLisp** extensibility and integration with other environments.

73 Description for kernel files:

`bignum.c` This file contains the bignum specific code.

`bitvect.c` This file contains the **`<simple-bit-vector>`** specific code.

`charutil.c` This file contains char internal routines. It mainly deal with UTF-8 encoding.

`class.c` This file contains the definition of **OpenLisp** Object system and generic functions.

`error.c` This file contains the condition system.

`function.c` This file contains the most standard functions as defined by ISLISP standard (**`cons, length, elt,`** ..). Functions are defined in the same order as in *ISO/IEC 13816:2007(E) ISLISP Programming Language* document.

`gc.c` This file contains the memory allocation scheme and the Garbage Collector.

`hash.c` This file implements hash-table functions.

`lap.c` This file contains the LAP code interpreter.

`memory.c` This file allocates memory and initializes **OpenLisp** memory zone. On some systems (mostly WIN32 and unix with mmap), it deals with the virtual memory manager to provide dynamic zone allocations and extensions.

misc.c This file implements **OpenLisp** extensions to ISLISP standard (**cadr**, **cddr**, **delete**, **reverse**...).

module.c This file implements the **OpenLisp** module facility (**require**, **provide**).

number.c This file implements integer, float and mixed arithmetic functions. When the C macro named **_IEEE31** is defined (default), **OpenLisp** uses a float representation based on 31 bits (or 63 bits) + 1 bit tag. That way, floats are not allocated and the float itself is coded in the address.

olinit.c This file is used to bootstrap OpenLisp, it allocates memory and create the standard symbols binding.

physio.c This file contains the only system dependent code. It mainly deals with low-level I/O and advanced system features. It mostly relies on ISO C and POSIX features but can also contains specific code that uses the system API (*i.e.* NT, System V).

packages.c This file contains the package management routines.

print.c This file implements ouput functions (mainly **format** and related functions).

rational.c This file contains the rational specific code.

read.c This file implements input functions (mainly **read** and related functions)

sockets.c This file implements, when available, sockets extensions to streams functions.

streams.c This file implements streams functions.

types.c This file implements type checking for standard Lisp objects.

eval.c This file implements the evaluator (mainly **eval** function).

openlisp.c This file is the standard entry point of **OpenLisp**. It contains **main** and launch **OpenLisp** and its toplevel loop.

toplevel.c This file implements the standard toplevel loop.

74 Complementary files:

The following files are not always needed when **OpenLisp** is used as a transparent Lisp engine used from C/C++ code.

debug.c This file implement the internal debugger.

evalbuf.c This file contains functions that help communication between C and Lisp.

odsp.c This file implements the OpenLisp Dynamic Server Page.

olsimple.c This file is an example how **OpenLisp** can be used from a standard C/C++ code. It works in character mode as well as with Windows API (Windows 3.x, Windows 9x or Windows NT).

reglisp.c This file implements regular expression functions.

xmllisp.c This file implements simple a xml lisp reader..

dos/graphics.c This file implements a minimal set of functions to display graphics on MS-DOS (now obsolete).

dos/dosterm.c This file implements the **virtty** functions for MS-DOS.

nt/ntterm.c This file implements the **virtty** functions for NT/9x.

win/windows.c This file implements the **virtty** and graphics functions for Windows API (16/32 bits).

win/winterm.c This file implements a graphic toplevel under Windows.

x11/x11.c This file implements a graphic functions for X11.

References

[ABE85] Abelson & Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, McGraw Hill Book Company, 1985.

[BOE95] Boehm, H., *Dynamic Memory Allocation and Garbage Collection,* Computers in Physics, 3, May/June 1995, pp. 297-303.

[C90] ISO/IEC 9899:1990, *Programming Language - C,* 1990.

[CHA85a] Jérôme Chailloux, *Le-Lisp version 15, le manuel de référence*; documentation INRIA, Février 1985.

[CHA85b] Jérôme Chailloux, *La machine virtuelle LLM3*, rapport technique N° 55, INRIA, Juin 1985.

[CPP98] ISO/IEC, *14882:1998(E) - Programming Language - C++.* September 1998.

[DEL91] Vincent Delacour, *Gestion mémoire automatique pour langages de programmation de haut niveau*, Thèse d'Université PARIS VI - 14 Juin 1991.

[GAB85] Richard P. Gabriel, *Performance and Evaluation of Lisp Systems*; Research Reports and Notes - Computer Systems Series, The MIT Press, 1985.

[I3E85] IEEE standard 754-1985, *IEEE standard for binary floating point arithmetic*. IEEE New York 1985.

[ILO92] ILOG, *Le-Lisp version 16, le manuel de référence*; Février 1992.

[ILO94] ILOG, *ILOG Talk, Reference Manual, Version 3.13,* 1995.

[ISA99] ISO, *IS0/IEC 10967-2:1998, Information technology – Language independant arithmetic – Part 2: Elementary numerical functions.* 1998.

[ISC87] ISO, *IS0 8859-1:1987, Information processing – 8 bit single-byte coded graphic character sets – Part 1: Latin alphabet N°1.* 1987.

[ISU93] ISO, *IS0/IEC 10646-1:1993, Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.* 1993.

[ISL97] ISO, *ISO/IEC 13816:1997(E) Programming Language ISLISP*. 1997.

[ISL07] ISO, *ISO/IEC 13816:2007(E) Programming Language ISLISP*. 2007.

[IZU98] N. Izumi and T. Ito: *Interpreter and Compiler of the ISO Standard Lisp ISLISP, Proc National Convention of Information Processing Society of Japan*, Vol.1, pp.323 – 324 (1998)

[JUL86] Christian Jullien, *Le-Cool : Un langage orienté objet à hiérarchie multiple pour la représentation des connaissances en Intelligence Artificielle.* Thèse de doctorat de 3ème cycle.

[JUL91] Christian Jullien, *MLisp 1.7, manuel de référence*. (unpublished).

[KEE89] Keen, Sonya E, *Object-Oriented Programming in Common Lisp : A Programmer's Guide to CLOS,* Addison-Wesley, 1989.

[MCA62] John McCarthy, *LISP 1.5 Programmer's Manual*, The Computation Center and Research Laboratory of Electronics. Massachusetts Institute of Technology, 1962.

[PIT01] Kent M. Pitman, *Condition Handling in the Lisp Language Family*, 2001 appears in Advances in Exception Handling Techniques.

[PEY87] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, 1987 Prentice-Hall International Series In Computer Science.

[POS1-90] ISO/IEC 9945-1:1990, Information technology - *Portable Operating System Interface (POSIX) - Part 1 : System Application Program Interface (API)* [C Language], 1990.

[POS2-93] ISO/IEC 9945-2:1993, Information technology - *Portable Operating System Interface (POSIX) - Part 2 : Shell and Utilities* [Volume I & II], 1993.

[PSG-95] ISO/IEC 9945-1g:1995, Information technology - *Portable Operating System Interface (POSIX) - Part xx : Protocol Independent Interface (PII)* [P1003.1g/D6.1], 1995.

[QUE92] Christian Queinnec, *Sémantique des dialectes de Lisp*, X & INRIA (non publié).

[QUE94] Christian Queinnec, *Les Langages Lisp*, InterEditions, 1994.

[SCH67] H. Schorr and W. M. Waite. *An efficient Machine-Indépendant Procedure for Garbage Collection in Various List Structures.* Communications of the ACM, 10(8) :501-506, 1997.

[SER94] Manuel Serrano. *Vers une compilation portable et performante des langages fonctionnels.* Thèse d'Université PARIS VI.

[SEG92] Robert SEDGEWICK. *Algorithms in C++*. Addison Wesley, 1992.

[SOK93] Microsoft. *SOCKETS REFERENCE MANUAL, VERSION 1.1*, 1992-1995.

[SPI90] Eric Spir. *Gestion dynamique de la mémoire dans les langages de programmation - application à Lisp*, 1990.

[STE90] Guy Lewis Steele Jr. *COMMON LISP: The Language*, Digital Press, second edition, 1990.

[VOC85] ISO/IEC 2382/15, *Data processing - Vocabulary - Part 15 : Programming languages*, First edition - 1985-11-01.

[W&H84] Winston & Horn. *Lisp (second edition).* Addison Wesley, 1984.

[X3J91] X3J13. *Working Draft: ANSI - Programming Language COMMON LISP*.

[YUA91] Taiichi Yusa & Alt., The Kernel Lisp Language for ISO Lisp Standardization - The Japanese Proposal, (unpublished).

General Index