XLISP-PLUS : Another Object-oriented Lisp

Version 2.1d , January 2, 1992

Tom Almy : toma@sail.labs.tek.com
Windows version : Gabor Paller , paller@evt.bme.hu

Table of Contents

Introduction

Introduction to the Window port

XLisp command loop
Break command loop
Data types
The evaluator
Hook functions
Lexical conventions
Readtables
Symbol case control
Lambda lists
Objects
Symbols

Evaluation functions
Symbol functions
Property list functions
Hash table functions
Array functions
Sequence functions
List functions
Destructive list functions
Arithmetic functions
Bitwise logical functions
String functions
Character functions
Structure functions
Object functions
Predicate functions
Control constructs
Looping constructs
The program feature
Input/output functions
The FORMAT function
File I/O functions
String stream functions
Debugging and error handling functions
System functions
Graphic functions

XLisp server

Additional functions and utilities

Introduction

XLISP-PLUS is an enhanced version of David Michael Betz's XLISP to have additional features of Common Lisp. XLISP-PLUS is distributed for the IBM-PC family and for UNIX, but can be easily ported to other platforms. Complete source code is provided (in "C") to allow easy modification and extension.

Since XLISP-PLUS is based on XLISP, most XLISP programs will run on XLISP-PLUS. Since XLISP-PLUS incorporates many more features of Common Lisp, many small Common Lisp applications will run on XLISP-PLUS with little modification.

Many Common Lisp functions are built into XLISP-PLUS. In addition , XLISP defines the objects 'Object' and 'Class' as primitives. 'Object' is the only class that has no superclass and hence is the root of the class heirarchy tree. 'Class' is the class of which all classes are instances (it is the only object that is an instance of itself).

This document is a brief description of XLISP-PLUS. It assumes some knowledge of LISP and some understanding of the concepts of object-oriented programming.

You will probably also need a copy of "Common Lisp: The Language" by Guy L. Steele, Jr., published by Digital Press to use as a reference for some of the Common Lisp functions that are described only briefly in this document.

XLISP-PLUS has a number of compilation options to eliminate groups of functions and to tailor itself to various environments. Unless otherwise indicated this manual assumes all options are enabled and the system dependent code is as complete as that provided for the MS/DOS environment. Assistance for using or porting XLISP-PLUS can be obtained on the USENET newsgroup comp.lang.lisp.x, or by writing to Tom Almy at the Internet address toma@sail.labs.tek.com. You can also reach Tom by writing to him at 17830 SW Shasta Trail, Tualatin, OR 97062, USA.

Introduction to the Windows port

Introduction to the Windows port

The Windows version of the XLisp is the unchanged XLisp version 2.1d adapted to the Windows 3.x environment. The XLisp features were preserved (except the graphics capability - see later) while the system now can take advantage of the Windows environment - multitasking , virtual memory , task-to-task communication. You need at least Windows 3.0 and a 286 machine to run this program.

When the XLisp for Windows is invoked , it generally does not offer more than the DOS versions , it has the same line editor although its window can be resized. The LOAD and RESTORE functions can also be issued by using the menu bar. Special Windows functions were not included into the language - instead the system offers a server function. The server interface is an easy-to-use subroutine library (DLL) by which client tasks can log in to the XLisp server , send tasks to it and get replies. XLisp can run in the background exploiting the idle time of the Windows or if the clients can stop for the processing time , run at full speed. See details about the communication DLL at the XLisp server chapter.

Unfortunately the WINDOWS.H file got stuck on the XLisp headers so some global names must have been modified. For this reason the Windows version is provided in itself not together with the other stuffs because almost all the sources have been changed a little.

XLisp for Windows was written by Gabor Paller (E-mail : paller@evt.bme.hu , Regular Mail : Department of Electromagnetic Theory , Technical University of Budapest , 18. Egry J. str. 1521 Budapest , Hungary) , questions about this versions should be adressed to me while questions about general XLisp should be sent to Thomas Almy (see Introduction).

XLisp command loop

When XLISP is started, it first tries to load the workspace "xlisp.wks", or an alternative file specified with the "-wfilename" option, from the current directory. If that file doesn't exist, or the "-w" flag is in the command line, XLISP builds an initial workspace, empty except for the built-in functions and symbols.

Then, providing no workspace file was loaded, XLISP attempts to load "init.lsp" from the current directory. It then loads any files named as parameters on the command line (after appending ".lsp" to their names). If the -s flag is in the command line , XLisp is started as a server task - this is generally done by the interface library (XSERVER.DLL). The XLisp server task is shown as an icon at the beginning although it can be maximized , in this case its window shows the requests and replies processed so far. XLISP then issues the following prompt.

>

This indicates that XLISP is waiting for an expression to be typed. The expression can come from the user (keyboard) or from a client task. If a client task is logged in you can still type in expressions from the keyboard but doing this you can corrupt the protocol built between the client and the server.

When a complete expression has been entered, XLISP attempts to evaluate that expression. If the expression evaluates successfully, XLISP prints the result and then returns for another expression.

The following control characters can be used while XLISP is waiting for input:

Backspace	delete last character
Del	delete last character
tab	tabs over (treated as space by XLISP reader)
ctrl-C	goto top level
ctrl-G	cleanup and return one level
ctrl-Z	end of file (returns one level or exits program)
ctrl-P	proceed (continue)
ctrl-T	print information (added function by TAA , now in window (PG))

Under Windows the following control characters can be typed while XLISP is executing (providing standard input has not been redirected away from the console):

ctrl-B	BREAK -- enter break loop
ctrl-S	Pause until another key is struck
ctrl-C	go to top level (if lucky: ctrl-B,ctrl-C is safer)
ctrl-T	print information

Break command loop

When XLISP encounters an error while evaluating an expression, it attempts to handle the error in the following way:

If the symbol *breakenable* is true, the message corresponding to the error is printed. If the error is correctable, the correction message is printed.

If the symbol *tracenable* is true, a trace back is printed. The number of entries printed depends on the value of the symbol *tracelimit*. If this symbol is set to something other than a number, the entire trace back stack is printed.

XLISP then enters a read/eval/print loop to allow the user to examine the state of the interpreter in the context of the error. This loop differs from the normal top-level read/eval/print loop in that if the user invokes the function 'continue', XLISP will continue from a correctable error. If the user invokes the function 'clean-up', XLISP will abort the break loop and return to the top level or the next lower numbered break loop. When in a break loop, XLISP prefixes the break level to the normal prompt.

If the symbol *breakenable* is NIL, XLISP looks for a surrounding errset function. If one is found, XLISP examines the value of the print flag. If this flag is true, the error message is printed. In any case, XLISP causes the errset function call to return NIL. If there is no surrounding errset function, XLISP prints the error message and returns to the top level.

Data types

There are several different data types available to XLISP-PLUS programmers. Typical implementation limits are shown for 32 bit word systems. Values in square brackets apply to 16 bit MS-DOS implementations.

All data nodes are effectively cons cells consisting of two pointers and and one or two bytes of identification flags (9 or 10 bytes per cell). Node space is managed and garbage collected by XLISP. Array and string storage is either allocated by the C runtime or managed and garbaged collected by XLISP (compilation option). If C does the allocation, memory fragmentation can occur. Fragmentation can be eliminated by saving the image and restarting XLISP-PLUS.

NIL
Lists
Arrays
Character strings
Symbols
Fixnums
Ratios
Characters
Floating point numbers
Complex numbers
Objects
Streams
String streams
Subrs
Fsubrs
Closures
Structures
Hash tables
Random states

NIL

Unlike the original XLISP, NIL is a symbol (although not in the *obarray*), to allowing setting its properties.

Lists

Either NIL or a CDR-linked list of cons cells, terminated by a symbol (typically NIL). Circular lists are allowable, but can cause problems with some functions so they must be used with care.

Arrays

The CDR field of an array points to the dynamically allocated data array, while the CAR contains the integer length of the array. Elements in the data array are pointers to other cells [Size limited to about 16360].

Character strings

Implemented like arrays, except string array is byte indexed and contains the actual characters. Note that unlike the underlying C, the null character (value 0) is valid. [Size limited to about 65500]

Symbols

Implemented as a 4 element array. The elements are value cell, function cell, property list, and print name (a character string node). Print names are limited to 100 characters. There are also flags for constant and special. Values bound to special symbols (declared with DEFVAR or DEFPARAMETER) are always dynamically bound, rather than being lexically bound.

Fixnums

Small integers (> -129 and <256) are statically allocated and are thus always EQ integers of the same value. The CAR field is used to hold the value, which is a 32 bit signed integer.

Ratios

The CAR field is used to hold the numerator while the CDR field is used to hold the denominator. The numerator is a 32 bit signed value while the denominator is a 31 bit positive value.

Characters

All characters are statically allocated and are thus EQ characters of the same value. The CAR field is used to hold the value. In XLISP characters are "unsigned" and thus range in value from 0 to 255.

Flonums (floating point numbers)

The CAR and CDR fields hold the value, which is typically a 64 bit IEEE floating point number.

Complex numbers

Part of the math extension compilation option. Internally implemented as an array of the real and imaginary parts. The parts can be either both fixnums or both flonums. Any function which would return an fixnum complex number with a zero imaginary part returns just the fixnum.

Objects

Implemented as an array of instance variable count plus one elements. The first element is the object's class, while the remaining arguments are the instance variables.

Streams

The CAR and CDR fields are used in a system dependent way as a file pointer.

String streams

Implemented as a tconc-style list of characters.

Subrs

The subrs are built-in functions. The CAR field points to the actual code to execute, while the CDR field is an internal pointer to the name of the function.

Fsubrs

The fsubrs are special forms. Same implementation as subrs.

Closures

The closures (user defined functions) are implemented as an array of 11 elements:

1. name symbol or NIL
2. 'lambda or 'macro
3. list of required arguments
4. optional arguments as list of (<arg> <init> <specified-p>) triples.
5. &rest argument
6. &key arguments as list of (<key> <arg> <init> <specified-p>) quadruples.
7. &aux arguments as list of (<arg> <init>) pairs.
8. function body
9. value environment
10. function environment
11. argument list (unprocessed)

Structures

Implemented as an array with first element being a pointer to the structure name string, and the remaining elements being the structure elements.

Hash tables

Implemented as a structure of varying length with no generalized accessing functions, but with a special print function (print functions not available for standard structures).

Random states

Implemented as a structure with a single element which is the random state (here a fixnum, but could change without impacting xlisp programs).

The evaluator

The process of evaluation in XLISP:

Strings, characters, numbers of any type, objects, arrays, structures, streams, subrs, fsubrs and closures evaluate to themselves.

Symbols act as variables and are evaluated by retrieving the value associated with their current binding.

Lists are evaluated by examining the first element of the list and then taking one of the following actions:

If it is a symbol, the functional binding of the symbol is retrieved.
If it is a lambda expression, a closure is constructed for the function described by the lambda expression.
If it is a subr, fsubr or closure, it stands for itself.
Any other value is an error.

Then, the value produced by the previous step is examined:

If it is a subr or closure, the remaining list elements are evaluated and the subr or closure is applied to these evaluated expressions.

If it is an fsubr, the fsubr is called with the remaining list elements as arguments (unevaluated).

If it is a macro, the macro is expanded with the remaining list elements as arguments (unevaluated). The macro expansion is then evaluated in place of the original macro call. If the symbol *displace-macros* is not NIL, then the expanded macro will (destructively) replace the original macro expression. This means that the macro will only be expanded once, but the original code will be lost. The displacement will not take place unless the macro expands into a list. The standard XLISP practice is the macro will be expanded each time the expression is evaluated, which negates some of the advantages of using macros.

Hook functions

The evalhook and applyhook facility are useful for implementing debugging programs or just observing the operation of XLISP. It is possible to control evaluation of forms in any context.

If the symbol *evalhook* is bound to a function closure, then every call of eval will call this function. The function takes two arguements, the form to be evaluated and execution environment. During the execution of this function, *evalhook* (and *applyhook*) are dynamically bound to NIL to prevent undesirable recursion. This "hook" function returns the result of the evaluation.

If the symbol *applyhook* is bound to a function, then every function application within an eval will call this function (note that the function apply, and others which do not use eval, will not invoke the apply hook function). The function takes two arguments, the function closure and the argument list (which is already evaluated). During execution of this hook function, *applyhook* (and *evalhook*) are dynamically bound to NIL to prevent undesired recursion. This function is to return the result of the function application.

Note that the hook functions cannot reset *evalhook* or *applyhook* to NIL, because upon exit these values will be reset. An excape mechanism is provided -- execution of 'top-level', or any error that causes return to the top level, will unhook the functions. Applications should bind these values either via progv, evalhook, or applyhook.

The functions 'evalhook' and 'applyhook' allowed for controlled application of the hook functions. The form supplied as an argument to 'evalhook', or the function application given to 'applyhook', are not hooked themselves, but any subsidiary forms and applications are. In addition, by supplying NIL values for the hook functions, 'evalhook' can be used to execute a form within a specific environment passed as an argument.

An additional hook function exists for the garbage collector. If the symbol *gc-hook* is bound to a function, then this function is called after every garbage collection. The function has two arguments. The first is the total number of nodes, and the second is the number of nodes free. The return value is ignored. During the execution of the function, *gc-hook* is dynamically bound to NIL to prevent undesirable recursion.

Lexical conventions

The following conventions must be followed when entering XLISP programs:

Comments in XLISP code begin with a semi-colon character and continue to the end of the line.

Except when escape sequences are used, symbol names in XLISP can consist of any sequence of non-blank printable characters except the terminating macro characters:

( ) ' ` , " ;

and the escape characters:

\ |

In addition, the first character may not be '#' (non-terminating macro character), nor may the symbol have identical syntax with a numeric literal. Uppercase and lowercase characters are not distinguished within symbol names because, by default, lowercase characters are mapped to uppercase on input.

Any printing character, including whitespace, may be part of a symbol name when escape characters are used. The backslash escapes the following character, while multiple characters can be escaped by placing them between vertical bars. At all times the backslash must be used to escape either escape characters.

For semantic reasons, certain chararacter sequences should/can never be used as symbols in XLISP. A single period is used to denote dotted lists. The symbol NIL represents an empty list. Symbols starting with a colon are keywords, and will always evaluate to themselves. Thus they should not be used as regular symbols. The symbol T is also reserved for use as the truth value.

Fixnum (integer) literals consist of a sequence of digits optionally beginning with a sign ('+' or '-'). The range of values an integer can represent is limited by the size of a C 'long' on the machine on which XLISP is running.

Ratio literals consist of two integer literals separated by a slash character ('/'). The second number, the denominator, must be positive. Ratios are automatically reduced to their cannonical form; if they are integral, then they are reduced to an integer.

Flonum (floating point) literals consist of a sequence of digits optionally beginning with a sign ('+' or '-') and including one or both of an embedded decimal point or a trailing exponent. The optional exponent is denoted by an 'E' or 'e' followed by an optional sign and one or more digits. The range of values a floating point number can represent is limited by the size of a C 'double' on most machines on which XLISP is running.

Numeric literals cannot have embedded escape characters. If they do, they are treated as symbols. Thus '12\3' is a symbol even though it would appear to be identical to '123'.

Complex literals are constructed using a read-macro of the format #C(r i), where r is the real part and i is the imaginary part. The numeric fields can be any valid fixnum, ratio, or flonum literal. If either field has a ratio or flonum literal, then both values are converted to flonums. Fixnum complex literals with a zero imaginary part are automatically reduced to fixnums.

Character literals are handled via the #\ read-macro construct:

#\<char> == the ASCII code of the printing character
#\newline == ASCII linefeed character

#\space == ASCII space character
#\rubout == ASCII rubout (DEL)
#\C-<char> == ASCII control character
#\M-<char> == ASCII character with msb set (Meta character)
#\M-C-<char> == ASCII control character with msb set

Literal strings are sequences of characters surrounded by double quotes (the " read-macro). Within quoted strings the '\' character is used to allow non-printable characters to be included. The codes recognized are:

\\	means the character '\'
\n	means newline
\t	means tab
\r	means return
\f	means form feed
\nnn	means the character whose octal code is nnn

Readtables

The behaviour of the reader is controlled by a data structure called a "readtable". The reader uses the symbol *readtable* to locate the current readtable. This table controls the interpretation of input characters -- if it is changed then the section Lexical conventions may not apply. The readtable is an array with 256 entries, one for each of the extended ASCII character codes. Each entry contains one of the following values, with the initial entries assigned to the values indicated:

:white-space	A whitespace character - tab, cr, lf, ff, space
(:tmacro . fun)	terminating readmacro - ( ) " , ; ' `
(:nmacro . fun)	non-terminating readmacro - #
:sescape	Single escape character - \
:mescape	Multiple escape character - |
:constituent	Indicating a symbol constituent (all printing characters not listed above)
NIL	Indicating an invalid character (everything else)

In the case of :TMACRO and :NMACRO, the "fun" component is a function. This can either be a built-in readmacro function or a lambda expression. The function takes two parameters. The first is the input stream and the second is the character that caused the invocation of the readmacro. The readmacro function should return NIL to indicate that the character should be treated as white space or a value consed with NIL to indicate that the readmacro should be treated as an occurance of the specified value. Of course, the readmacro code is free to read additional characters from the input stream. A :nmacro is a symbol constituent except as the first character of a symbol. As an example, the following read macro allows the square brackets to be used as a more visibly appealing alternative to the SEND function:

```
(setf (aref *readtable* (char-int #\[)) ; #\[ table entry
      (cons :tmacro
            (lambda (f c &aux ex) ; second arg is not used
                  (do ()
                      ((eq (peek-char t f) #\]))
                      (setf ex (append ex (list (read f)))))
                  (read-char f) ; toss the trailing #\]
                  (cons (cons 'send ex) NIL))))

(setf (aref *readtable* (char-int #\]))
      (cons :tmacro
            (lambda (f c)
                  (error "misplaced right bracket"))))
```

XLISP defines several useful read macros:

```
'<expr> == (quote <expr>)
`<expr> == (backquote <expr>)
,<expr> == (comma <expr>)
,@<expr>          == (comma-at <expr>)
#'<expr>          == (function <expr>)
#(<expr>...)      == an array of the specified expressions
#S(<structtype> [<slotname> <value>]...)
                  ==  structure of specified type and initial values
#.<expr>          == result of evaluating <expr>
#x<hdigits>       == a hexadecimal number (0-9,A-F)
#o<odigits>       == an octal number (0-7)
#b<bdigits>       == a binary number (0-1)
#|  |#            == a comment
#:<symbol>        == an uninterned symbol
```

```
#C(r i)              == a complex number
```

Symbol case control

XLISP-PLUS uses two variables, *READTABLE-CASE* and *PRINT-CASE* to determine case conversion during reading and printing of symbols. *READTABLE-CASE* can have the values :UPCASE :DOWNCASE :PRESERVE or :INVERT, while *PRINT-CASE* can have the values :UPCASE or :DOWNCASE. By default, or when other values have been specified, both are :UPCASE.

When *READTABLE-CASE* is :UPCASE, all unescaped lowercase characters are converted to uppercase when read. When it is :DOWNCASE, all unescaped uppercase characters are converted to lowercase. This mode is not very useful because the predefined symbols are all uppercase and would need to be escaped to read them. When *READTABLE-CASE* is :PRESERVE, no conversion takes place. This allows case sensitive input with predefined functions in uppercase. The final choice, :INVERT, will invert the case of any symbol that is not mixed case. This provides case sensitive input while making the predefined functions and variables appear to be in lowercase.

The printing of symbols involves the settings of both *READTABLE-CASE* and *PRINT-CASE*. When *READTABLE-CASE* is :UPCASE, lowercase characters are escaped (unless PRINC is used), and uppercase characters are printed in the case specified by *PRINT-CASE*. When *READTABLE-CASE* is :DOWNCASE, uppercase characters are escaped (unless PRINC is used), and lowercase are printed in the case specified by *PRINT-CASE*. The remaining *READTABLE-CASE* modes ignore *PRINT-CASE* and do not escape alphabetic characters. :PRESERVE never changes the case of characters while :INVERT inverts the case of any non mixed-case symbols.

There are four major useful combinations of these modes:

A: *READTABLE-CASE* :UPCASE *PRINT-CASE* :UPCASE

"Traditional" mode. Case insensitive input; must escape to put lowercase characters in symbol names. Symbols print exactly as they are stored, with lowercase characters escaped when PRIN1 is used.

B: *READTABLE-CASE* :UPCASE *PRINT-CASE* :DOWNCASE

"Eyesaver" mode. Case insensitive input; must escape to put lowercase characters in symbol name. Symbols print entirely in lowercase except symbols escaped when lowercase characters present with PRIN1.

C: *READTABLE-CASE* :PRESERVE

"Oldfashioned case sensitive" mode. Case sensitive input. Predefined symbols must be typed in uppercase. No alpha quoting needed. Symbols print exactly as stored.

D: *READTABLE-CASE* :INVERT

"Modern case sensitive" mode. Case sensitive input. Predefined symbols must be typed in lowercase. Alpha quoting should be avoided. Predefined symbols print in lower case, other symbols print as they were entered.

As far as compatibility between these modes are concerned, data printed in mode A can be read in A, B, or C. Data printed in mode B can be read in A, B, and D. Data printed in mode C can be read in mode C, and if no lowercase symbols in modes A and B as well. Data printed in mode D can be read in mode D, and if no (internally) lowercase symbols in modes A and B as well. In addition, symbols containing characters requiring quoting are compatible among all modes.

Lambda lists

There are several forms in XLISP that require that a "lambda list" be specified. A lambda list is a definition of the arguments accepted by a function. There are four different types of arguments.

The lambda list starts with required arguments. Required arguments must be specified in every call to the function.

The required arguments are followed by the &optional arguments. Optional arguments may be provided or omitted in a call. An initialization expression may be specified to provide a default value for an &optional argument if it is omitted from a call. If no initialization expression is specified, an omitted argument is initialized to NIL. It is also possible to provide the name of a 'supplied-p' variable that can be used to determine if a call provided a value for the argument or if the initialization expression was used. If specified, the supplied-p variable will be bound to T if a value was specified in the call and NIL if the default value was used.

The &optional arguments are followed by the &rest argument. The &rest argument gets bound to the remainder of the argument list after the required and &optional arguments have been removed.

The &rest argument is followed by the &key arguments. When a keyword argument is passed to a function, a pair of values appears in the argument list. The first expression in the pair should evaluate to a keyword symbol (a symbol that begins with a ':'). The value of the second expression is the value of the keyword argument. Like &optional arguments, &key arguments can have initialization expressions and supplied-p variables. In addition, it is possible to specify the keyword to be used in a function call. If no keyword is specified, the keyword obtained by adding a ':' to the beginning of the keyword argument symbol is used. In other words, if the keyword argument symbol is 'foo', the keyword will be ':foo'. Extra keywords will signal an error unless &allow-other-keys is present, in which case the extra keywords are ignored. In XLISP, the &allow-other-keys argument is ignored, and extra keywords are ignored.

The &key arguments are followed by the &aux variables. These are local variables that are bound during the evaluation of the function body. It is possible to have initialization expressions for the &aux variables. Here is the complete syntax for lambda lists:

```
(<rarg>...
[&optional [<oarg> | (<oarg> [<init> [<svar>]])]...]
[&rest <rarg>]
[&key
[<karg> |  ([<karg> |  (<key> <karg>)] [<init>  [<svar>]])] ...
[&allow-other-keys]]
[&aux [<aux> | (<aux> [<init>])]...])
```

where:

```
<rarg>  is a required argument symbol
<oarg>  is an &optional argument symbol
<rarg>  is the &rest argument symbol
<karg>  is a &key argument symbol
<key>   is a keyword symbol (starts with ':')
<aux>   is an auxiliary variable symbol
<init>          is an initialization expression
<svar>  is a supplied-p variable symbol
```

Objects

Definitions:

selector - a symbol used to select an appropriate method
message - a selector and a list of actual arguments
method - the code that implements a message

Since XLISP was created to provide a simple basis for experimenting with object-oriented programming, one of the primitive data types included is 'object'. In XLISP, an object consists of a data structure containing a pointer to the object's class as well as an array containing the values of the object's instance variables.

Officially, there is no way to see inside an object (look at the values of its instance variables). The only way to communicate with an object is by sending it a message.

You can send a message to an object using the 'send' function. This function takes the object as its first argument, the message selector as its second argument (which must be a symbol) and the message arguments as its remaining arguments.

The 'send' function determines the class of the receiving object and attempts to find a method corresponding to the message selector in the set of messages defined for that class. If the message is not found in the object's class and the class has a super-class, the search continues by looking at the messages defined for the super-class. This process continues from one super-class to the next until a method for the message is found. If no method is found, an error occurs.

To perform a method lookup starting with the method's superclass rather than the object's class, use the function 'send-super'. This allows a subclass to invoke a standard method in its parent class even though it overrides that method with its own specialized version.

When a method is found, the evaluator binds the receiving object to the symbol 'self' and evaluates the method using the remaining elements of the original list as arguments to the method. These arguments are always evaluated prior to being bound to their corresponding formal arguments. The result of evaluating the method becomes the result of the expression.

Two objects, both classes, are predefined: Object and Class. Both Object and Class are of class Class. The superclass of Class is Object, while Object has no superclass. Typical use is to create new classes (by sending :new to Class) to represent application objects. Objects of these classes, created by sending :new to the appropriate new class, are subclasses of Object. The Object method :show can be used to view the contents of any object.

The 'Object' class

Object : the top of the class hierarchy
Messages of the Object class

The 'Class' class

Class : The class of all object classes (including itself)
Messages of the Class class
Instance variables of the Class class

When a new instance of a class is created by sending the message ':new' to an existing class, the message ':isnew' followed by whatever parameters were passed to the ':new' message is sent to the newly created object. Therefore, when a new class is created by sending ':new' to class 'Class' the message ':isnew' is sent to Class automatically. To create a new class, a function of the following

format is used:

(setq <newclassname> (send Class :new <ivars> [<cvars> [<super>]]))

When a new class is created, an optional parameter may be specified indicating the superclass of the new class. If this parameter is omitted, the new class will be a subclass of 'Object'. A class inherits all instance variables, and methods from its super-class. Only class variables of a method's class are accessable.

Messages of the Object class

:show
Show an object's instance variables.

:class
Return the class of an object

:prin1 [<stream>]
Print the object. <stream> default, or NIL, is *standard-output*, T is *terminal-io*. Returns the object

:isnew
The default object initialization routine. Returns the object

:superclass
Get the superclass of the object. Returns NIL. (Defined in classes.lsp, see :superclass below)

:ismemberof <class>
Class membership. <class> : class name. Returns T if object member of class, else NIL (defined in classes.lsp)

:iskindof <class>
Class membership. <class> : class name. Returns T if object member of class or subclass of class, else NIL (defined in classes.lsp)

:respondsto <sel>
Selector knowledge. <sel> : message selector. Returns T if object responds to message selector, else NIL. (defined in classes.lsp)

:storeon
Read representation. Returns a list, that when executed will create a copy of the object. Only works for members of classes created with defclass. (defined in classes.lsp)

Messages of the Class class

:new

Create a new instance of a class. Returns the new class object

:isnew <ivars> [<cvars> [<super>]]

Initialize a new class. <ivars> : the list of instance variable symbol , <cvars>: the list of class variable symbols , <super> : the superclass (default is Object). Returns the new class object

:answer <msg> <fargs> <code>

Add a message to a class. <msg> : the message symbol , <fargs> : the formal argument list (lambda list) , <code> : a list of executable expressions. Returns the object

:superclass

Get the superclass of the object. Returns the superclass of the class. (defined in classes.lsp)

:messages

Get the list of messages of the class. Returns association list of message selectors and closures for messages. (defined in classes.lsp)

:storeon

Read representation. Returns a list, that when executed will re-create the class and its methods. (defined in classes.lsp)

Instance variables of 'CLASS' class

MESSAGES
An association list of message names and closures implementing the messages.

IVARS
List of names of instance variables.

CVARS
List of names of class variables.

CVAL
Array of class variable values.

SUPERCLASS
The superclass of this class or NIL if no superclass (only for class OBJECT).

IVARCNT
Instance variables in this class (length of IVARS)

IVARTOTAL
Total instance variables for this class and all superclasses of this class.

PNAME
Printname string for this class.

Symbols

All values are initially NIL unless otherwise specified. All are special variables unless indicated to be constants.

NIL
Represents empty list and the boolean value for "false". The value of NIL is NIL, and cannot be changed (it is a constant). (car NIL) and (cdr NIL) are also defined to be NIL.

t
Boolean value "true" is constant with value t.

self
Within a method context, the current object otherwise initially unbound.

object
Constant, value is the class 'Object'.'

class
Constant, value is the class 'Class'.

internal-time-units-per-second
Integer constant to divide returned times by to get time in seconds.

pi
Floating point aproximation of pi (constant defined when math extension is compiled).

****obarray****
The object hash table. Length of array is a compilation option. Objects are hashed using the hash function and are placed on a list in the appropriate array slot.

****terminal-io****
Stream bound to keyboard and display. Do not alter.

****standard-input****
The standard input stream, initially stdin.

****standard-output****
The standard output stream, initially stdout.

****error-output****
The error output stream (used by all error messages), initially same as *terminal-io*.

****trace-output****
The trace output stream (used by the trace function), initially same as *terminal-io*.

****debug-io****
The break loop i/o stream, initially same as *terminal-io*. System messages (other than error messages) also print out on this stream.

****breakenable****
Flag controlling entering break loop on errors

****tracelist****
List of names of functions to trace, as set by trace function.

****tracenable****
Enable trace back printout on errors.

****tracelimit****
Number of levels of trace back information.

****evalhook****
User substitute for the evaluator function

****applyhook****
User substitute for function application.

****readtable****
The current readtable.

****unbound****
Indicator for unbound symbols. A constant. Do not use this symbol since accessing any variable to which this has been bound will cause an unbound symbol error message.

****gc-flag****
Controls the printing of gc messages. When non-NIL, a message is printed after each garbage collection giving the total number of nodes and the number of nodes free.

gc-hook
Function to call after garbage collection
integer-format
Format for printing integers (when not bound to a string, defaults to "%d" or "%ld" depending on implementation)
ratio-format
Format for printing ratios (when not bound to a string, defaults to "%d/%d" or "%ld/%ld" depending on implementation)
float-format
Format for printing floats (when not bound to a string, defaults to "%g")
readtable-case
Symbol read and output case.
print-case
Symbol output case when printing.
print-level
When bound to a number, list levels beyond this value are printed as '#'. Used by all printing functions. Good precaution to avoid getting caught in circular lists.
print-length
When bound to a number, lists longer than this value are printed as '...'. Used by all printing functions. Good precaution to avoid getting caught in circular lists.
displace-macros
When not NIL, macros are replaced by their expansions when exectuted.
random-state
The default random-state used by the random function.

There are several symbols maintained by the read/eval/print loop. The symbols '+', '++', and '+++' are bound to the most recent three input expressions. The symbols '*', '**' and '***' are bound to the most recent three results. The symbol '-' is bound to the expression currently being evaluated. It becomes the value of '+' at the end of the evaluation.

Evaluation functions

eval
apply
funcall
quote
function
identity
backquote
comma
comma-at
lambda
get-lambda-expression
macroexpand
macroexpand-1

Eval

Syntax : (eval <expr>)

Evaluate an XLisp expression
<expr> the expression to be evaluated

Returns : The result of evaluating the expression

Apply

Syntax : (apply <fun> <arg>...<args>)

Apply a function to a list of arguments

<fun>	The function to apply (or function symbol). May not be macro or fsubr.
<arg>	Initial arguments, which are CONSed to...
<args>	The argument list

Returns : The result of applying the function to the arguments

Funcall

Syntax : (funcall <fun> <arg>...)

Call a function with arguments

<fun>	The function to call (or function symbol). May not be macro or fsubr.
<arg>	Arguments to pass to the function

Returns : The result of calling the function with the arguments

Quote

Syntax : (quote <expr>)

Return an expression unevaluated
fsubr
<expr> The expression to be quoted (quoted)

Returns : <expr> unevaluated

Function

Syntax : (function <expr>)

Get the functional interpretation

subr		
expr>		The symbol or lambda expression (quoted)

Returns : the functional interpretation

Identity

Syntax : (identity <expr>)

Return the expression. New function. In common.lsp
<expr> The expression

Returns : the expression

Backquote

Syntax : (backquote <expr>)

Fill in a template. fsubr. Note: an improved backquote facility, which works properly when nested, is available by loading the file backquot.lsp.

<expr> The template (quoted)

Returns : a copy of the template with comma and comma-at expressions expanded.

Comma

Syntax : (comma <expr>)

Comma expression. (Never executed) As the object of a backquote expansion, the expression is evaluated and becomes an object in the enclosing list.

Comma-at

Syntax : (comma-at <expr>)

Comma-at expression. (Never executed) As the object of a backquote expansion, the expression is evaluated (and must evaluate to a list) and is then spliced into the enclosing list.

Lambda

Syntax : (lambda <args> <expr>...)

Make a function closure. fsubr

<args> Formal argument list (lambda list) (quoted)
<expr> Expressions of the function body (quoted)

Returns : the function closure

Get-lambda-expression

Syntax : (get-lambda-expression <closure>)

Get the lambda expression
<closure> The closure

Returns : the original lambda expression

Macroexpand

Syntax : (macroexpand <form>)

Recursively expand macro calls
<form> The form to expand

Returns : the macro expansion

Macroexpand-1

Syntax : (macroexpand-1 <form>)

Expand a macro call
<form> The macro call form

Returns : the macro expansion

Symbol functions

set
setq
psetq
setf
defsetf
push
pushnew
pop
incf
decf
defun
defmacro
gensym
intern
make-symbol
symbol-name
symbol-value
symbol-function
symbol-plist
hash
makunbound
fmakunbound
unintern
defconstant
defparameter
defvar

Set

Syntax : (set <sym> <expr>)

Set the global value of a symbol
<sym> The symbol being set
<expr> The new value

Returns : the new value

Setq

Syntax : (setq [<sym> <expr>]...)

Set the value of a symbol. fsubr

<sym> The symbol being set (quoted)
<expr> The new value

Returns : the new value

Psetq

Syntax : (psetq [<sym> <expr>]...)

Parallel version of setq. fsubr. All expressions are evaluated before any assignments are made.
<sym> the symbol being set (quoted)
<expr> the new value

Returns : the new value

Setf

Syntax : (setf [<place> <expr>]...)

Set the value of a field. fsubr.

<place> the field specifier (if a macro it is expanded, then the form arguments are evaluated):

- <sym> set value of a symbol
- (car <expr>) set car of a cons node
- (cdr <expr>) set cdr of a cons node
- (nth <n> <expr>) set nth car of a list
- (aref <expr> <n>) set nth element of an array or string
- (elt <expr> <n>) set nth element of a sequence
- (get <sym> <prop>) set value of a property
- (symbol-value <sym>) set global value of a symbol
- (symbol-function <sym>) set functional value of a symbol
- (symbol-plist <sym>) set property list of a symbol
- (gethash <key> <tbl> <def>) add or replace hash table entry. <def> is ignored
- (send <obj> :<ivar>) (When classes.lsp used), set instance variable of object.
- (<sym>-<element> <struct>) set the element of structure struct, type sym.
- (<fieldsym> <args>) the function stored in property *setf* in symbol <fieldsym> is applied to (<args> <expr>)

<value> the new value

Returns : the new value

Defsetf

Syntax : (defsetf <sym> <fcn>)

Define a setf field specifier

Syntax : (defsetf <sym> <fargs> (<value>) <expr>...)

Defined as macro in common.lsp. Convenient, Common Lisp compatible alternative to setting *setf* property directly, although second format is not as efficient.

<sym> field specifier symbol (quoted)
<fcn> function to use (quoted symbol) which takes the same arguments as the field specifier plus an additional argument for the value. The value must be returned.
<fargs> formal argument list (lambda list) (quoted)
<value> symbol bound to value to store (quoted).
<expr> expressions to evaluate (quoted). The last expression must return <value>.

Returns : the field specifier symbol

Push

Syntax : (push <expr> <place>)

Cons to a field. Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-NIL for best performance.

<place>field specifier being modified (see setf)
<expr> value to cons to field

Returns : the new value which is (CONS <expr> <place>)

Pushnew

Syntax : (pushnew <expr> <place> &key :test :test-not :key)

Cons new to a field. Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-NIL for best performance.

<place>	field specifier being modified (see setf)
<expr>	value to cons to field, if not already MEMBER of field
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function list argument (defaults to identity)

Returns : the new value which is (CONS <expr> <place>) or <place>

Pop

Syntax : (pop <place>)

Remove first element of a field. Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-NIL for best performance.

<place>the field being modified (see setf)

Returns : (CAR <place>), field changed to (CDR <place>)

Incf , decf

Syntax : (incf <place> [<value>])

Increment a field

Syntax : (decf <place> [<value>])

Decrement a field
Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-NIL for best performance.

<place>field specifier being modified (see setf)
<value>Numeric value (default 1)

Returns : the new value which is (+ <place> <value>) or (- <place> <value>)

Defun , defmacro

Syntax : (defun <sym> <fargs> <expr>...)

Define a function

Syntax : (defmacro <sym> <fargs> <expr>...)

Define a macro. fsubr

<sym> symbol being defined (quoted)
<fargs> formal argument list (lambda list) (quoted)
<expr> expressions the body of the function (quoted)

Returns : the function symbol

Gensym

Syntax : (gensym [<tag>])

Generate a symbol.

<tag>	string or number

Returns : the new symbol, uninterned

Intern

Syntax : (intern <pname>)

Make an interned symbol.
<pname> the symbol's print name string

Returns : the new symbol

Make-symbol

Syntax : (make-symbol <pname>)

Make an uninterned symbol
<pname> the symbol's print name string

Returns : the new symbol

Symbol-name

Syntax : (symbol-name <sym>)

Get the print name of a symbol
<sym> the symbol

Returns : the symbol's print name

Symbol-value

Syntax : (symbol-value <sym>)

Get the value of a symbol
<sym> the symbol

Returns : the symbol's value

Symbol-function

Syntax : (symbol-function <sym>)

Get the functional value of a symbol
<sym> the symbol

Returns : the symbol's functional value

Symbol-plist

Syntax : (symbol-plist <sym>)

Get the property list of a symbol
<sym> the symbol

Returns : the symbol's property list

Hash

Syntax : (hash <expr> <n>)

Compute the hash index
<expr> the object to hash
<n> the table size (positive integer)

Returns : the hash index (integer 0 to n-1)

Makunbound

Syntax : (makunbound <sym>)

Make a symbol value be unbound. You cannot unbind constants.
<sym> the symbol

Returns : the symbol

Fmakunbound

Syntax : (fmakunbound <sym>)

Make a symbol function be unbound. Defined in init.lsp
<sym> the symbol

Returns : the symbol

Unintern

Syntax : (unintern <sym>)

Unintern a symbol. Defined in common.lsp
<sym> the symbol

Returns : t if successful, NIL if symbol not interned

Defconstant

Syntax : (defconstant <sym> <val>)

Define a constant. fsubr.
<sym> the symbol
<val> the value

Returns : the value

Defparameter

Syntax : (defparameter <sym> <val>)

Define a parameter. fsubr.
<sym> the symbol
<val> the value

Returns : the value

Defvar

Syntax : (defvar <sym> [<val>])

Define a variable. fsubr. Variable only initialized if not previously defined.

<sym> the symbol
<val> the initial value, or NIL if absent.

Returns : the current value

Property list functions

Note that property names are not limited to symbols.

get
putprop
remprop

Get

Syntax : (get <sym> <prop>)

Get the value of a property
<sym> the symbol
<prop> the property symbol

Returns : the property value or NIL

Putprop

Syntax : (putprop <sym> <val> <prop>)

Put a property onto a property list
<sym> the symbol
<val> the property value
<prop> the property symbol

Returns : the property value

Remprop

Syntax : (remprop <sym> <prop>)

Delete a property
<sym> the symbol
<prop> the property symbol

Returns : NIL

Hash table functions

A hash table is implemented as an structure of type hash-table. No general accessing functions are provided, and hash tables print out using the angle bracket convention (not readable by READ). The first element is the comparison function. The remaining elements contain association lists of keys (that hash to the same value) and their data.

make-hash-table
gethash
remhash
clrhash
hash-table-count
maphash

Make-hash-table

Syntax : (make-hash-table &key :size :test)

Make a hash table

:size	size of hash table -- should be a prime number. Default is 31.
:test	comparison function. Defaults to eql.

Returns : the hash table

Gethash

Syntax : (gethash <key> <table> [<def>])

Extract from hash table. See also gethash in SETF.

<key> hash key
<table> hash table
<def> value to return on no match (default is NIL)

Returns : associated data, if found, or <def> if not found.

Remhash

Syntax : (remhash <key> <table>)

Delete from hash table
<key> hash key
<table> hash table

Returns : T if deleted, NIL if not in table

Clrhash

Syntax : (clrhash <table>)

Clear the hash table
<table> hash table

Returns : NIL, all entries cleared from table

Hash-table-count

Syntax : (hash-table-count <table>)

Number of entries in hash table
<table> hash table

Returns : integer number of entries in table

Maphash

Syntax : (maphash <fcn> <table>)

Map function over table entries

<fcn> the function or function name, a function of two arguments, the first is bound to the key, and the second the value of each table entry in turn.

<table> hash table

Returns : NIL

Array functions

Note that sequence functions also work on arrays.

aref
make-array
vector

Aref

Syntax : (aref <array> <n>)

Get the nth element of an array. See setf for setting elements of arrays
<array> the array (or string)
<n> the array index (integer, zero based)

Returns : the value of the array element

Make-array

Syntax : (make-array <size>)

Make a new array
<size> the size of the new array (integer)

Returns : the new array

Vector

Syntax : (vector <expr>...)

Make an initialized vector
<expr> the vector elements

Returns : the new vector

Sequence functions

These functions work on sequences - lists, arrays, or strings.

concatenate
elt
map
every
notevery
some
notany
length
reverse
nreverse
subseq
search
remove
remove-if
remove-if-not
count-if
find-if
position-if
delete
delete-if
delete-if-not
reduce
remove-duplicates
fill
replace

Concatenate

Syntax : (concatenate <type> <expr> ...)

Concatenate sequences. If result type is string, sequences must contain only characters.

<type> result type, one of CONS, LIST, ARRAY, or STRING
<expr> zero or more sequences to concatenate

Returns : a sequence which is the concatenation of the argument sequences

Elt

Syntax : (elt <expr> <n>)

Get the nth element of a sequence
<expr> the sequence
<n> the index of element to return

Returns : the element if the index is in bounds, otherwise error

Map

Syntax : (map <type> <fcn> <expr> ...)

Apply function to successive elements
<type> result type, one of CONS, LIST, ARRAY, STRING, or NIL
<fcn> the function or function name
<expr> a sequence for each argument of the function

Returns : a new sequence of type <type>.

Every,Notevery

Syntax : (every <fcn> <expr> ...)

Syntax : (notevery <fcn> <expr> ...)

Apply function to elements until false

<fcn> the function or function name

<expr> a sequence for each argument of the function

Returns : every returns last evaluated function result , notevery returns T if there is a NIL function result, else NIL

Some,notany

Syntax : (some <fcn> <expr> ...)

Syntax : (notany <fcn> <expr> ...)

Apply function to elements until true

<fcn> the function or function name
<expr> a sequence for each argument of the function

Returns : some returns first non-NIL function result, or NIL notany returns NIL if there is a non-NIL function result, else T

Length

Syntax : (length <expr>)

Find the length of a sequence
<expr> the list, vector or string

Returns : the length of the list, vector or string

Reverse,nreverse

Syntax : (reverse <expr>)

Reverse a sequence

Syntax : (nreverse <expr>)

Destructively reverse a sequence
<expr> the sequence to reverse

Returns : a new sequence in the reverse order

Subseq

Syntax : (subseq <seq> <start> [<end>])

Extract a subsequence
<seq> the sequence
<start> the starting position (zero origin)
<end> the ending position + 1 (defaults to end) or NIL for end of sequence

Returns : the sequence between <start> and <end>

Search

Syntax : (search <seq1> <seq2> &key :test :test-not :key :start1 :end1 :start2 :end2)

Search for sequence

<seq1> the sequence to search for
<seq2> the sequence to search in

:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function arguments (defaults to identity)
:start1	starting index in <seq1>
:end1	index of end+1 in <seq1> or NIL for end of sequence
:start2	starting index in <seq2>
:end2	index of end+1 in <seq2> or NIL for end of sequence

Returns : position of first match

Remove

Syntax : (remove <expr> <seq> &key :test :test-not :key :start :end)

Remove elements from a sequence

<expr>	the element to remove
<seq>	the sequence
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function sequence argument (defaults to identity)
:start	starting index
:end	index of end+1, or NIL for (length <seq>)

Returns : copy of sequence with matching expressions removed

Remove-if,remove-if-not

Syntax : (remove-if <test> <seq> &key :key :start :end)

Remove elements that pass test

Syntax : (remove-if-not <test> <seq> &key :key :start :end)

remove elements that fail test

<test>	the test predicate		
<seq>	the sequence		
:key	function to	apply to test	function argument (defaults to identity)
:start	starting index		
:end	index of end+1, or NIL for (length <seq>)		

Returns : copy of sequence with matching or non-matching elements removed

Count-if

Syntax : (count-if <test> <seq> &key :key :start :end)

Count elements that pass test

<test>	the test predicate	
<seq>	the sequence	
:key	function to apply to test function	argument (defaults to identity)
:start	starting index	
:end	index of end+1, or NIL for (length <seq>)	

Returns : count of matching elements

Find-if

Syntax : (find-if <test> <seq> &key :key :start :end)

Find first element that passes test

<test>	the test predicate	
<seq>	the list	
:key		function to apply to test function argument (defaults to identity)
:start		starting index
:end		index of end+1, or NIL for (length <seq>)

Returns : first element of sequence that passes test

Position-if

Syntax : (position-if <test> <seq> &key :key :start :end)

Find position of first element that passes test

<test>	the test predicate	
<seq>	the list	
:key	function to apply to	test function argument (defaults to identity)
:start	starting index	
:end	index of end+1, or NIL for (length <seq>)	

Returns : position of first element of sequence that passes test , or NIL.

Delete

Syntax : (delete <expr> <seq> &key :key :test :test-not :start :end)

Delete elements from a sequence

<expr>	the element to delete
<seq>	the sequence
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function sequence argument (defaults to identity)
:start	starting index
:end	index of end+1, or NIL for (length <seq>)

Returns : the sequence with the matching expressions deleted

Delete-if,delete-if-not

Syntax : (delete-if <test> <seq> &key :key :start :end)

Delete elements that pass test

Syntax : (delete-if-not <test> <seq> &key :key :start :end)

Delete elements that fail test

<test>	the test predicate
<seq>	the sequence
:key	function to apply to test function argument (defaults to identity)
:start	starting index
:end	index of end+1, or NIL for (length <seq>)

Returns : the sequence with matching or non-matching elements deleted

Reduce

Syntax : (reduce <fcn> <seq> &key :initial-value :start :end)

Reduce sequence to single value

<fcn>	function (of two arguments) to apply to result of previous function application (or first element) and each member of sequence.
<seq>	the sequence
:initial-value	value to use as first argument in first function application rather than using the first element of the sequence.
:start	starting index
:end	index of end+1, or NIL for (length <seq>)

Returns : if sequence is empty and there is no initial value, returns result of applying function to zero arguements. If there is a single element, returns the element. Otherwise returns the result of the last function application.

Remove-duplicates

Syntax : (remove-duplicates <seq> &key :test :test-not :key :start :end)

Remove duplicates from sequence

<seq>	the sequence
:test	comparison function (default eql)
:test-not	comparison function (sense inverted)
:key	function to apply to test function arguments (defaults to identity)
:start	starting index
:end	index of end+1, or NIL for (length <seq>)

Returns : copy of sequence with duplicates removed.

Fill

Syntax : (fill <seq> <expr> &key :start :end)

Replace items in sequence. Defined in common.lsp

<seq>	the sequence
<expr>	new value to place in sequence
:start	starting index
:end	index of end+1, or NIL for (length <seq>)

Returns : sequence with items replaced with new item

Replace

Syntax : (replace <seq1> <seq2> &key :start1 :end1 :start2 :end2)

Replace items in sequence from sequence. Defined in common.lsp

<seq1>	the sequence to modify
<seq2>	sequence with new items
:start1	starting index in <seq1>
:end1	index of end+1 in <seq1> or NIL for end of sequence
:start2	starting index in <seq2>
:end2	index of end+1 in <seq2> or NIL for end of sequence

Returns : first sequence with items replaced

List functions

car
cdr
cxxr,cxxxr,cxxxxr
first
rest
second
third
fourth
cons
acons
list
list*
append
last
butlast
nth
nthcdr
member
assoc
mapc
mapcar
mapl
maplist
mapcan
mapcon
subst
sublis
pairlis
copy-list
copy-alist
copy-tree
intersection
union
set-difference
set-exclusive-or
nintersection
nunion
nset-difference
nset-exclusive-or
adjoin

Car

Syntax : (car <expr>)

Return the car of a list node. The ***first*** function is the synonym of car.
<expr> the list node

Returns : the car of the list node

Cdr

Syntax : (cdr <exp>)

Return the cdr of a list node. The ***rest*** function is the synonym of cdr.
<expr> the list node

Returns : the cdr of the list node

Cxxr,cxxxr,cxxxxr

Syntax : (cxxr <expr>)

All cxxr combinations

Syntax : (cxxxr <expr>)

All cxxxr combinations

Syntax : (cxxxxr <expr>)

All cxxxxr combinations

Synonyms :

(second <expr>) a synonym for cadr
(third <expr>) a synonym for caddr
(fourth <expr>) a synonym for cadddr

Cons

Syntax : (cons <expr1> <expr2>)

Construct a new list node

<expr1>	the car of the new list node
<expr2>	the cdr of the new list node

Returns : the new list node

Acons

Syntax : (acons <expr1> <expr2> <alist>)

Add to front of assoc list. Defined in common.lsp
<expr1> key of new association
<expr2> value of new association
<alist> association list
Returns : new association list, which is (cons (cons <expr1> <expr2>) <expr3>))

*List,list**

Syntax : (list <expr>...)
Syntax : (list* <expr> ... <list>)

Create a list of values
<expr> expressions to be combined into a list

Returns : the new list

Append

Syntax : (append <expr>...)

Append lists
<expr> lists whose elements are to be appended

Returns : the new list

Last

Syntax : (last <list>)

Return the last list node of a list
<list> the list

Returns : the last list node in the list

Butlast

Syntax : (butlast <list> [<n>])

Return copy of all but last of list

<list>	the list
<n>	count of elements to omit (default 1)

Returns : copy of list with last element(s) absent.

Nth

Syntax : (nth <n> <list>)

Return the nth element of a list

<n>	the number of the element to return (zero origin)
<list>	the list

Returns : the nth element or NIL if the list isn't that long

Nthcdr

Syntax : (nthcdr <n> <list>)

Return the nth cdr of a list

<n>	the number of the element to return (zero origin)
<list>	the list

Returns : the nth cdr or NIL if the list isn't that long

Member

Syntax : (member <expr> <list> &key :test :test-not :key)

Find an expression in a list

<expr>	the expression to find
<list>	the list to search
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function list argument. (defaults to identity)

Returns : the remainder of the list starting with the expression

Assoc

Syntax : (assoc <expr> <alist> &key :test :test-not :key)

Find an expression in an A-list

<expr>	the expression to find
<alist>	the association list
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function list argument (defaults to identity)

Returns : the alist entry or NIL

Mapc

Syntax : (mapc <fcn> <list1> <list>...)

Apply function to successive cars

<fcn> the function or function name
<listn> a list for each argument of the function

Returns : the first list of arguments

Mapcar

Syntax : (mapcar <fcn> <list1> <list>...)

Apply function to successive cars
<fcn> the function or function name
<listn> a list for each argument of the function

Returns : a list of the values returned

Mapl

Syntax : (mapl <fcn> <list1> <list>...)

Apply function to successive cdrs

<fcn>	the function or function name
<listn>	a list for each argument of the function

Returns : the first list of arguments

Maplist

Syntax : (maplist <fcn> <list1> <list>...)

Apply function to successive cdrs

<fcn> the function or function name
<listn> a list for each argument of the function

Returns : a list of the values returned

Mapcan

Syntax : (mapcan <fcn> <list1> <list>...)

Apply function to successive cars
<fcn> the function or function name
<listn> a list for each argument of the function

Returns : list of return values nconc'd together

Mapcon

Syntax : (mapcon <fcn> <list1> <list>...)

Apply function to successive cdrs
<fcn> the function or function name
<listn> a list for each argument of the function

Returns : list of return values nconc'd together

Subst

Syntax : (subst <to> <from> <expr> &key :test :test-not :key)

Substitute expressions. Does minimum copying as required by Common Lisp

<to>	the new expression
<from>	the old expression
<expr>	the expression in which to do the substitutions
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function expression argument (defaults to identity)

Returns : the expression with substitutions

Sublis

Syntax : (sublis <alist> <expr> &key :test :test-not :key)

Substitute with an A-list. Does minimum copying as required by Common Lisp

<alist>	the association list
<expr>	the expression in which to do the substitutions
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function expression argument (defaults to identity)

Returns : the expression with substitutions

Pairlis

Syntax : (pairlis <keys> <values> [<alist>])

Build an A-list from two lists. In file common.lsp
<keys> list of association keys
<values> list of association values, same length as keys
<alist> existing association list, default NIL

Returns : new association list

Copy-list

Syntax : (copy-list <list>)

Copy the top level of a list. In file common.lsp
<list> the list

Returns : a copy of the list (new cons cells in top level)

Copy-alist

Syntax : (copy-alist <alist>)

Copy an association list. In file common.lsp
<alist> the association list

Returns : a copy of the association list (keys and values not copies)

Copy-tree

Syntax : (copy-tree <tree>)

Copy a tree. In file common.lsp
<tree> a tree structure of cons cells

Returns : a copy of the tree structure

Set functions

Syntax :

(intersection <list1> <list2> &key :test :test-not :key)
(union <list1> <list2> &key :test :test-not :key)
(set-difference <list1> <list2> &key :test :test-not :key)
(set-exclusive-or <list1> <list2> &key :test :test-not :key)
(nintersection <list1> <list2> &key :test :test-not :key)
(nunion <list1> <list2> &key :test :test-not :key)
(nset-difference <list1> <list2> &key :test :test-not :key)
(nset-exclusive-or <list1> <list2> &key :test :test-not :key)

set-exclusive-or and nset-exclusive-or defined in common.lsp. nunion, nintersection, and nset-difference are aliased to their non-destructive counterparts in common.lsp.

<list1>	first list
<list2>	second list
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function arguments (defaults to identity)

Returns : intersection: list of all elements in both lists
union: list of all elements in either list
set-diference: list of all elements in first list but not in second list
set-exclusive-or: list of all elements in only one list
"n" versions are potentially destructive.

Adjoin

Syntax : (adjoin <expr> <list> :test :test-not :key)

Add unique to list

<expr>	new element to add
<list>	the list
:test	the test function (defaults to eql)
:test-not	the test function <sense inverted)
:key	function to apply to test function arguments (defaults to identity)

Returns : if element not in list then (cons <expr> <list>), else <list>.

Destructive list functions

See also nreverse, delete, delete-if, delete-if-not, fill, and replace under Sequence functions, setf under Symbol functions, and nintersection, nunion, nset-difference, and nset-exclusive-or under List functions.

rplaca
rplacd
nconc
sort

Rplaca

Syntax : (rplaca <list> <expr>)

Replace the car of a list node
<list> the list node
<expr> the new value for the car of the list node

Returns : the list node after updating the car

Rplacd

Syntax : (rplacd <list> <expr>)

Replace the cdr of a list node
<list> the list node
<expr> the new value for the cdr of the list node

Returns : the list node after updating the cdr

Nconc

Syntax : (nconc <list>...)

Destructively concatenate lists
<list> lists to concatenate

Returns : the result of concatenating the lists

Sort

Syntax : (sort <list> <test> &key :key)

Sort a list

<list>	the list to sort
<test>	the comparison function
:key	function to apply to comparison function arguments (defaults to identity)

Returns : the sorted list

Arithmetic functions

Warning: integer and ratio calculations that overflow give erroneous results. On systems with IEEE floating point, the values +INF and -INF result from overflowing floating point calculations.

The math extension option adds complex numbers, ratios, new functions, and additional functionality to some existing functions. Because of the size of the extension, and the performance loss it entails, some users may not wish to include it. This section documents the math functions both with and without the extension.

Functions that are described as having floating point arguments (SIN COS TAN ASIN ACOS ATAN EXPT EXP SQRT) will take arguments of any type (real or complex) when the math extension is used. In the descriptions, "rational number" means integer or ratio only, and "real number" means floating point number or rational only.

Any rational results are reduced to canonical form (the gcd of the numerator and denominator is 1, the denominator is positive); integral results are reduced to integers. Integer complex numbers with zero imaginary parts are reduced to integers.

truncate
round
floor
ceiling
float
+
-
*
/
1+
1-
rem
mod
min
max
abs
signum
gcd
lcm
random
make-random-state
sin
cos
tan
asin
acos
atan
sinh
cosh
tanh
asinh
acosh
atanh
expt
exp
cis
log

sqrt
numerator
denominator
complex
realpart
imagpart
conjugate
phase
≤
<=
=
/=
>=
≥

Truncate , round , floor , ceiling

Syntax : (truncate <expr> <denom>)

Truncates toward zero

Syntax : (round <expr> <denom>)

Rounds toward nearest integer

Syntax : (floor <expr> <denom>)

Truncates toward negative infinity

Syntax : (ceiling <expr> <denom>)

Truncates toward infinity

Round, floor, and ceiling, and the second argument of truncate, are part of the math extension. Results too big to be represented as integers are returned as floating point numbers as part of the math extension. Integers are returned as is.

<expr> the real number
<denom> real number to divide <expr> by before converting

Returns : the integer result of converting the number

Float

Syntax : (float <expr>)

Converts an integer to a floating point number
<expr> the real number

Returns : the number as a floating point number

+

Syntax : (+ [<expr>...])

Add a list of numbers

With no arguments returns addition identity, 0 (integer)
<expr> the numbers

Returns : the result of the addition

-

Syntax : (- <expr>...)

Subtract a list of numbers or negate a single number
<expr> the numbers

Returns : the result of the subtraction

*

Syntax : (* [<expr>...])

Multiply a list of numbers. With no arguments returns multiplication identity, 1

<expr> the numbers

Returns : the result of the multiplication

/

Syntax : (/ <expr>...)

Divide a list of numbers or invert a single number. With the math extension, division of integer numbers results in a rational quotient, rather than integer. To perform integer division, use TRUNCATE. If an integer complex is divided by an integer, the quotient is floating point complex.

<expr> the numbers

Returns : the result of the division

1+

Syntax : (1+ <expr>)

Add one to a number
<expr> the number

Returns : the number plus one

1-

Syntax : (1- <expr>)

Subtract one from a number
<expr> the number

Returns : the number minus one

Rem

Syntax : (rem <expr>...)

Remainder of a list of numbers. With the math extension, only two arguments allowed.
<expr> the real numbers (must be integers, without math extension)

Returns : the result of the remainder operation (remainder with truncating division)

Mod

Syntax : (mod <expr1> <expr2>)

Number modulo another number. Part of math extension.

<expr1>	real number
<expr2>	real number divisor (may not be zero)

Returns : the remainder after dividing <expr1> by <expr2> using flooring division, thus there is no discontinuity in the function around zero.

Min

Syntax : (min <expr>...)

The smallest of a list of numbers
<expr> the real numbers

Returns : the smallest number in the list

Max

Syntax : (max <expr>...)

The largest of a list of numbers
<expr> the real numbers

Returns : the largest number in the list

Abs

Syntax : (abs <expr>)

The absolute value of a number.
<expr> the number

Returns : the absolute value of the number, which is the floating point magnitude for complex numbers.

Signum

Syntax : (signum <expr>)

Get the sign of a number. Defined in common.lsp
<expr> the number

Returns : zero if number is zero, one if positive, or negative one if negative. Numeric type is same as number. For a complex number, returns unit magnitude but same phase as number.

Gcd

Syntax : (gcd [<n>...])

Compute the greatest common divisor. With no arguments returns 0, with one argument returns the argument.

<n>	The number(s) (integer)

Returns : the greatest common divisor

Lcm

Syntax : (lcm <n>...)

Compute the least common multiple. Part of math extension.

<n>	The number(s) (integer)

Returns : the least common multiple

Random

Syntax : (random <n> [<state>])

Compute a pseudo-random number
<n> the real number upper bound
<state> a random-state (default is *random-state*)

Returns : a random number in range [0,n)

Make-random-state

Syntax : (make-random-state [<state>])

Create a random state

<state> a random-state, t, or NIL (default NIL). NIL means *random-state*

Returns : If <state> is t, a random random-state, otherwise a copy of <state>

Sin , cos , tan , asin , acos

Syntax : (sin <expr>)

Compute the sine of a number

Syntax : (cos <expr>)

Compute the cosine of a number

Syntax : (tan <expr>)

Compute the tangent of a number

Syntax : (asin <expr>)

Compute the arc sine of a number

Syntax : (acos <expr>)

Compute the arc cosine of a number
<expr> the floating point number

Returns : the sine, cosine, tangent, arc sine, or arc cosine of the number

Atan

Syntax : (atan <expr> [<expr2>])

Compute the arc tangent of a number
<expr> the floating point number (numerator)
<expr2> the denominator, default 1. May only be specified if math extension installed

Returns : the arc tangent of <expr>/<expr2>

Sinh , cosh , tanh , asinh , acosh , atanh

Syntax : (sinh <expr>)

Compute the hyperbolic sine of a number

Syntax : (cosh <expr>)

Compute the hyperbolic cosine of a number

Syntax : (tanh <expr>)

Compute the hyperbolic tangent of a number

Syntax : (asinh <expr>)

Compute the hyperbolic arc sine of a number

Syntax : (acosh <expr>)

Compute the hyperbolic arc cosine of a number

Syntax : (atanh <expr>)

Compute the hyperbolic arc tangent of a number. Defined in common.lsp
<expr> the number

Returns : the hyperbolic sine, cosine, tangent, arc sine, arc cosine, or arc tangent of the number.

Expt

Syntax : (expt <x-expr> <y-expr>)

Compute X to the Y power

<x-expr>	the number
<y-expr>	the exponent

Returns : x to the y power. If y is a fixnum, then the result type is the same as the type of x, unless fixnum or ratio and it would overflow, then the result type is a flonum.

Exp

Syntax : (exp <x-expr>)

Compute E to the X power
<x-expr> the floating point number

Returns : e to the x power

Cis

Syntax : (cis <x-expr>)

Compute cosine + I sine. Defined in common.lsp
<x-expr> the number

Returns : e to the ix power

Log

Syntax : (log <expr> [<base>])

Compute the logarithm. Part of the math extension
<expr> the number
<base> the base, default is e

Returns : log base <base> of <expr>

Sqrt

Syntax : (sqrt <expr>)

Compute the square root of a number
<expr> the number

Returns : the square root of the number

Numerator

Syntax : (numerator <expr>)

Get the numerator of a number. Part of math extension
<expr> rational number

Returns : numerator of number (number if integer)

Denominator

Syntax : (denominator <expr>)

Get the denominator of a number. Part of math extension
<expr> rational number

Returns : denominator of number (1 if integer)

Complex

Syntax : (complex <real> [<imag>])

Convert to complex number. Part of math extension.
<real> real number real part
<imag> real number imaginary part (default 0)

Returns : the complex number

Realpart

Syntax : (realpart <expr>)

Get the real part of a number. Part of the math extension
<expr> the number

Returns : the real part of a complex number, or the number itself if a real number

Imagpart

Syntax : (imagpart <expr>)

Get the imaginary part of a number. Part of the math extension
<expr> the number

Returns : the imaginary part of a complex number, or zero of the type of the number if a real number.

Conjugate

Syntax : (conjugate <expr>)

Get the conjugate of a number. Part of the math extension
<expr> the number

Returns : the conjugate of a complex number, or the number itself if a real number.

Phase

Syntax : (phase <expr>)

Get the phase of a number. Part of the math extension
<expr> the number

Returns : the phase angle, equivalent to (atan (imagpart <expr>) (realpart <expr>))

< , <= , = , /= , >= , >

Syntax : (< <n1> <n2>...)

Test for less than

Syntax : (<= <n1> <n2>...)

Test for less than or equal to

Syntax : (= <n1> <n2>...)

Test for equal to

Syntax : (/= <n1> <n2>...)

Test for not equal to

Syntax : (>= <n1> <n2>...)

Test for greater than or equal to

Syntax : (> <n1> <n2>...)

Test for greater than

<n1>	the first real number to compare
<n2>	the second real number to compare

Returns : the result of comparing <n1> with <n2>...

Bitwise logical functions

logand
logior
logxor
lognot
logtest
ash

Logand

Syntax : (logand [<expr>...])

The bitwise and of a list of integers. With no arguments returns identity -1
<expr> the integers

Returns : the result of the and operation

Logior

Syntax : (logior [<expr>...])

The bitwise inclusive or of a list of integers. With no arguments returns identity 0
<expr> the integers

Returns : the result of the inclusive or operation

Logxor

Syntax : (logxor [<expr>...])

The bitwise exclusive or of a list of integers. With no arguments returns identity 0
<expr> the integers

Returns : the result of the exclusive or operation

Lognot

Syntax : (lognot <expr>)

The bitwise not of an integer
<expr> the integer

Returns : the bitwise inversion of integer

Logtest

Syntax : (logtest <expr1> <expr2>)

Test bitwise and of two integers. Defined in common.lsp

<expr1>	the first integer
<expr2>	the second integer

Returns : T if the result of the and operation is non-zero, else NIL

Ash

Syntax : (ash <expr1> <expr2>)

Arithmetic shift. Part of math extension

<expr1>	integer to shift
<expr2>	number of bit positions to shift (positive is to left)

Returns : shifted integer

String functions

Note: functions with names starting "string" will also accept a symbol, in which case the symbol's print name is used.

string
string-trim
string-left-trim
string-right-trim
string-upcase
string-downcase
nstring-upcase
nstring-downcase
strcat
string<
string<=
string=
string/=
string>=
string>
string_lessp
string-not-greaterp
string-equal
string-not-equal
string-not-lessp
string-greaterp

String

Syntax : (string <expr>)

Make a string from an integer ASCII value

<expr>	an integer (which is first converted into its ASCII character value), string, character, or symbol

Returns : the string representation of the argument

String-trim

Syntax : (string-trim <bag> <str>)

Trim both ends of a string
<bag> a string containing characters to trim
<str> the string to trim

Returns : a trimed copy of the string

String-left-trim

Syntax : (string-left-trim <bag> <str>)

Trim the left end of a string
<bag> a string containing characters to trim
<str> the string to trim

Returns : a trimed copy of the string

String-right-trim

Syntax : (string-right-trim <bag> <str>)

Trim the right end of a string
<bag> a string containing characters to trim
<str> the string to trim

Returns : a trimed copy of the string

String-upcase

Syntax : (string-upcase <str> &key :start :end)

Convert to uppercase

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or NIL for end of string

Returns : a converted copy of the string

String-downcase

Syntax : (string-downcase <str> &key :start :end)

Convert to lowercase

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or NIL for end of string

Returns : a converted copy of the string

Nstring-upcase

Syntax : (nstring-upcase <str> &key :start :end)

Convert to uppercase

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or NIL for end of string

Returns : the converted string (not a copy)

Nstring-downcase

Syntax : (nstring-downcase <str> &key :start :end)

Convert to lowercase

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or NIL for end of string

Returns : the converted string (not a copy)

Strcat

Syntax : (strcat <expr>...)

Concatenate strings. Macro in init.lsp, to maintain compatibility with XLISP. See CONCATENATE for preferred function.

<expr> the strings to concatenate

Returns : the result of concatenating the strings

String<,string<=,string=,string/=,string>=,string>

Syntax : (string< <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string<= <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string= <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string/= <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string>= <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string> <str1> <str2> &key :start1 :end1 :start2 :end2)

<str1>	the first string to compare
<str2>	the second string to compare
:start1	first substring starting offset
:end1	first substring ending offset + 1 or NIL for end of string
:start2	second substring starting offset
:end2	second substring ending offset + 1 or NIL for end of string

Returns : string=: t if predicate is true, NIL otherwise others: If predicate is true then number of initial matching characters, else NIL.
Note: case is significant with these comparison functions.

String-lessp,string-not-greaterp,string-equal,string-not-equal,string-not-lessp,string-greaterp

Syntax : (string-lessp <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string-not-greaterp <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string-equal <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string-not-equal <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string-not-lessp <str1> <str2> &key :start1 :end1 :start2 :end2)

Syntax : (string-greaterp <str1> <str2> &key :start1 :end1 :start2 :end2)

<str1>	the first string to compare
<str2>	the second string to compare
:start1	first substring starting offset
:end1	first substring ending offset + 1 or NIL for end of string
:start2	second substring starting offset
:end2	second substring ending offset + 1 or NIL for end of string

Returns : string-equal: t if predicate is true, NIL otherwise. others: If predicate is true then number of initial matching characters, else NIL
Note: case is not significant with these comparison functions.

Character functions

- char
- upper-case-p
- lower-case-p
- both-case-p
- digit-char-p
- char-code
- code-char
- char-upcase
- char-downcase
- digit-char
- char-int
- int-char
- char<
- char<=
- char=
- char/=
- char>=
- char>
- char-lessp
- char-not-greaterp
- char-equal
- char-not-equal
- char-not-lessp
- char-greaterp

Char

Syntax : (char <string> <index>)

Extract a character from a string
<string> the string
<index>the string index (zero relative)

Returns : the ascii code of the character

Upper-case-p

Syntax : (upper-case-p <chr>)

Is this an upper case character ?
<chr> the character

Returns : true if the character is upper case, NIL otherwise

Lower-case-p

Syntax : (lower-case-p <chr>)

Is this a lower case character ?
<chr> the character

Returns : true if the character is lower case, NIL otherwise

Both-case-p

Syntax : (both-case-p <chr>)

Is this an alphabetic (either case) character ?
<chr> the character

Returns : true if the character is alphabetic, NIL otherwise

Digit-char-p

Syntax : (digit-char-p <chr>)

Is this a digit character ?
<chr> the character

Returns : the digit weight if character is a digit, NIL otherwise

Char-code

Syntax : (char-code <chr>)

Get the ASCII code of a character
<chr> the character

Returns : the ASCII character code (integer, parity bit stripped)

Code-char

Syntax : (code-char <code>)

Get the character with a specified ASCII code
<code> the ASCII code (integer, range 0-127)

Returns : the character with that code or NIL

Char-upcase

Syntax : (char-upcase <chr>)

Convert a character to upper case
<chr> the character

Returns : the upper case character

Char-downcase

Syntax : (char-downcase <chr>)

Convert a character to lower case
<chr> the character

Returns : the lower case character

Digit-char

Syntax : (digit-char <n>)

Convertt a digit weight to a digit
<n> the digit weight (integer)

Returns : the digit character or NIL

Char-int

Syntax : (char-int <chr>)

Convert a character to an integer
<chr> the character

Returns : the ASCII character code (range 0-255)

Int-char

Syntax : (int-char <int>)

Convert an integer to a character
<int> the ASCII character code (treated modulo 256)

Returns : the character with that code

Char<,char<=,char=,char/=,char>=,char>

Syntax : (char< <chr1> <chr2>...)

Syntax : (char<= <chr1> <chr2>...)

Syntax : (char= <chr1> <chr2>...)

Syntax : (char/= <chr1> <chr2>...)

Syntax : (char>= <chr1> <chr2>...)

Syntax : (char> <chr1> <chr2>...)
<chr1> the first character to compare
<chr2> the second character(s) to compare

Returns : t if predicate is true, NIL otherwise
Note: case is significant with these comparison functions.

Char-lessp,char-not-greaterp,char-equal,char-not-equal,char-not-lessp,char-greaterp

Syntax : (char-lessp <chr1> <chr2>...)

Syntax : (char-not-greaterp <chr1> <chr2>...)

Syntax : (char-equal <chr1> <chr2>...)

Syntax : (char-not-equal <chr1> <chr2>...)

Syntax : (char-not-lessp <chr1> <chr2>...)

Syntax : (char-greaterp <chr1> <chr2>...)
<chr1> the first string to compare
<chr2> the second string(s) to compare

Returns : t if predicate is true, NIL otherwise
Note: case is not significant with these comparison functions.

Structure functions

XLISP provides a subset of the Common Lisp structure definition facility. No slot options are allowed, but slots can have default initialization expressions.

Syntax : (defstruct name <slot-desc>...)
or
(defstruct (name <option>...) <slot-desc>...)
fsubr

<name>	the structure name symbol (quoted)
<option>	option description (quoted)
<slot-desc>	slot descriptions (quoted)

Returns : the structure name

The recognized options are:

(:conc-name name)
(:include name [<slot-desc>...])

Note that if :CONC-NAME appears, it should be before :INCLUDE.

Each slot description takes the form:

<name>
or
(<name> <defexpr>)

If the default initialization expression is not specified, the slot will be initialized to NIL if no keyword argument is passed to the creation function.

DEFSTRUCT causes access functions to be created for each of the slots and also arranges that SETF will work with those access functions. The access function names are constructed by taking the structure name, appending a '-' and then appending the slot name. This can be overridden by using the :CONC-NAME option. DEFSTRUCT also makes a creation function called MAKE-<structname>, a copy function called COPY-<structname> and a predicate function called <structname>-P. The creation function takes keyword arguments for each of the slots. Structures can be created using the #S(read macro, as well.

The property *struct-slots* is added to the symbol that names the structure. This property consists of an association list of slot names and closures that evaluate to the initial values (NIL if no initial value expression).

For instance:

(defstruct foo bar (gag 2))

creates the following functions:

(foo-bar <expr>)
(setf (foo-bar <expr>) <value>)
(foo-gag <expr>)
(setf (foo-gag <expr>) <value>)
(make-foo &key :bar :gag)
(copy-foo <expr>)
(foo-p <expr>)

Object functions

Note that the functions provided in classes.lsp are useful but not necessary.

Messages defined for Object and Class are listed at Objects topic.

send
send-super
defclass
defmethod
definst

Send

Syntax : (send <object> <message> [<args>...])

Send a message
<object> the object to receive the message
<message> message sent to object
<args> arguments to method (if any)

Returns : the result of the method

Send-super

Syntax : (send-super <message> [<args>])

Send a message to superclass. Valid only in method context
<message> message sent to method's superclass
<args> arguments to method (if any)

Returns : the result of the method

Defclass

Syntax : (defclass <sym> <ivars> [<cvars> [<super>]])

Define a new class. Defined in class.lsp as a macro

<sym> symbol whose value is to be bound to the class object (quoted)

<ivars> list of instance variables (quoted). Instance variables specified either as <ivar> or (<ivar> <init>) to specify non-NIL default initial value.

<cvars>list of class variables (quoted)

<super> superclass, or Object if absent.

This function sends :SET-PNAME (defined in classes.lsp) to the new class to set the class' print name instance variable. Methods defined for classes defined with defclass:

(send <object> :<ivar>)

Returns the specified instance variable

(send <object> :SET-IVAR <ivar> <value>)

Used to set an instance variable, typically with setf.

(send <sym> :NEW {:<ivar> <init>})

Actually definition for :ISNEW. Creates new object initializing instance variables as specified in keyword arguments, or to their default if keyword argument is missing. Returns the object.

Defmethod

Syntax : (defmethod <class> <sym> <fargs> <expr> ...)

Define a new method. Defined in class.lsp as a macro.

<class> Class which will respond to message
<sym> Message name (quoted)
<fargs> Formal argument list. Leading "self" is implied (quoted)
<expr> Expressions constituting body of method (quoted)

Returns : the class object.

Definst

Syntax : (definst <class> <sym> [<args>...])

Define a new global instance. Defined in class.lsp as a macro

<class> Class of new object
<sym> Symbol whose value will be set to new object
<args> Arguments passed to :NEW (typically initial values for instance variables)

Predicate functions

atom
symbolp
numberp
null
not
listp
endp
consp
constantp
integerp
floatp
rationalp
complexp
stringp
characterp
arrayp
streamp
open-stream-p
input-stream-p
output-stream-p
objectp
classp
boundp
fboundp
functionp
minusp
zerop
plusp
evenp
oddp
subsetp
eq
eql
equal
equalp
typep

Atom

Syntax : (atom <expr>)

Is this an atom ?
<expr> the expression to check

Returns : t if the value is an atom, NIL otherwise

Symbolp

Syntax : (symbolp <expr>)

Is this a symbol ?
<expr> the expression to check

Returns : t if the expression is a symbol, NIL otherwise

Numberp

Syntax : (numberp <expr>)

Is this a number ?
<expr> the expression to check

Returns : t if the expression is a number, NIL otherwise

Null

Syntax : (null <expr>)

Is this an empty list ?
<expr> the list to check

Returns : t if the list is empty, NIL otherwise

Not

Syntax : (not <expr>)

Is this false ?

<expr> the expression to check

Return : t if the value is NIL, NIL otherwise

Listp

Syntax : (listp <expr>)

Is this a list ?
<expr> the expression to check

Returns : t if the value is a cons or NIL, NIL otherwise

Endp

Syntax : (endp <list>)

Is this the end of a list ?
<list> the list

Returns : t if the value is NIL, NIL otherwise

Consp

Syntax : (consp <expr>)

Is this a non-empty list ?
<expr> the expression to check

Returns : t if the value is a cons, NIL otherwise

Constantp

Syntax : (constantp <expr>)

Is this a constant ?
<expr> the expression to check

Returns : t if the value is a constant (basically, would EVAL <expr> repeatedly return the same thing ?), NIL otherwise.

Integerp

Syntax : (integerp <expr>)

Is this an integer ?
<expr> the expression to check

Returns : t if the value is an integer, NIL otherwise

Floatp

Syntax : (floatp <expr>)

Is this a float ?
<expr> the expression to check

Returns : t if the value is a float, NIL otherwise

Rationalp

Syntax : (rationalp <expr>)

Is this a rational number ? Part of math extension.
<expr> the expression to check

Returns : t if the value is rational (integer or ratio), NIL otherwise

Complexp

Syntax : (complexp <expr>)

Is this a complex number ? Part of math extension.
<expr> the expression to check

Returns : t if the value is a complex number, NIL otherwise

Stringp

Syntax : (stringp <expr>)

Is this a string ?
<expr> the expression to check

Returns : t if the value is a string, NIL otherwise

Characterp

Syntax : (characterp <expr>)

Is this a charactet ?
<expr> the expression to check

Returns : t if the value is a character, NIL otherwise

Arrayp

Syntax : (arrayp <expr>)

Is this an array ?
<expr> the expression to check

Returns : t if the value is an array, NIL otherwise

Streamp

Syntax : (streamp <expr>)

Is this a stream ?
<expr> the expression to check

Returns : t if the value is a stream, NIL otherwise

Open-stream-p

Syntax : (open-stream-p <stream>)

Is stream open ?
<stream> the stream

Returns : t if the stream is open, NIL otherwise

Input-stream-p

Syntax : (input-stream-p <stream>)

Is stream readable ?
<stream> the stream

Returns : t if stream is readable, NIL otherwise

Output-stream-p

Syntax : (output-stream-p <stream>)

Is stream writable ?
<stream> the stream

Returns : t if stream is writable, NIL otherwise

Objectp

Syntax : (objectp <expr>)

Is this an object ?
<expr> the expression to check

Returns : t if the value is an object, NIL otherwise

Classp

Syntax : (classp <expr>)

Is this a class object ?
<expr> the expression to check

Returns : t if the value is a class object, NIL otherwise

Boundp

Syntax : (boundp <sym>)

Is a value bound to this symbol ?
<sym> the symbol

Returns : t if a value is bound to the symbol, NIL otherwise

Fboundp

Syntax : (fboundp <sym>)

Is a functional value bound to this symbol ?
<sym> the symbol

Returns : t if a functional value is bound to the symbol, NIL otherwise

Functionp

Syntax : (functionp <sym>)

Is this a function ? Defined in common.lsp
<expr> the expression to check

Returns : t if the value is a function -- that is, can it be applied to arguments. This is true for any symbol (even those with no function binding), list with car being lambda, a closure, or subr. Otherwise returns NIL.

Minusp

Syntax : (minusp <expr>)

Is this number negative ?
<expr> the number to test

Returns : t if the number is negative, NIL otherwise

Zerop

Syntax : (zerop <expr>)

Is this number zero ?
<expr> the number to test

Returns : t if the number is zero, NIL otherwise

Plusp

Syntax : (plusp <expr>)

Is this number positive ?
<expr> the number to test

Returns : t if the number is positive, NIL otherwise

Evenp

Syntax : (evenp <expr>)

Is this integer even ?
<expr> the integer to test

Returns : t if the integer is even, NIL otherwise

Oddp

Syntax : (oddp <expr>)

Is this integer odd ?
<expr> the integer to test

Returns : t if the integer is odd, NIL otherwise

Subsetp

Syntax : (subsetp <list1> <list2> &key :test :test-not :key)

Is set a subset ?

<list1>	the first list
<list2>	the second list
:test	test function (defaults to eql)
:test-not	test function (sense inverted)
:key	function to apply to test function arguments (defaults to identity)

Returns : t if every element of the first list is in the second list, NIL otherwise

Eq,eql,equal,equalp

Syntax : (eq <expr1> <expr2>)

Syntax : (eql <expr1> <expr2>)

Syntax : (equal <expr1> <expr2>)

Syntax : (equalp <expr1> <expr2>)

Are the expressions equal ? equalp defined in common.lsp

<expr1>	the first expression
<expr2>	the second expression

Returns : t if equal, NIL otherwise. Each is progressively more liberal in what is "equal":
eq : identical pointers -- works with characters , symbols, and arbitrarily small integers
eql : works with all , if same type (see also ≡)
equal : lists and strings
equalp : case insensitive characters (and strings), numbers of differing types, arrays (which can be equalp to string containing same elements)

Typep

Syntax : (typep <expr> <type>)

Is this a specified type ?

<expr> the expression to test

<type> the type specifier. Symbols can either be one of those listed under type-of or one of:

ATOM	any atom
NULL	NIL
LIST	matches NIL or any cons cell
STREAM	any stream
NUMBER	any number type
RATIONAL	fixnum or ratio (math extension)
STRUCT	any structure (except hash-table)
FUNCTION	any function, as defined by functionp

The specifer can also be a form (which can be nested). All form elements are quoted. Valid form cars:

or	any of the cdr type specifiers must be true
and	all of the cdr type specifiers must be true
not	the single cdr type specifier must be false

satisfies the result of applying the cdr predicate function to <expr>

member	<expr> must be eql to one of the cdr values
object	<expr> must be an object, of class specified by the single cdr value. The cdr value can be a symbol which must evaluate to a class.

Note that everything is of type T, and nothing is of type NIL.

Returns : t if <expr> is of type <type>, NIL otherwise.

Control constructs

cond
and
or
if
when
unless
case
let
let*
flet
catch
throw
unwind-protect

Cond

Syntax : (cond <pair>...)

Evaluate conditionally. fsubr

<pair> pair consisting of:
(<pred> <expr>...)
where
<pred> is a predicate expression
<expr> evaluated if the predicate is not NIL

Returns : the value of the first expression whose predicate is not NIL

And

Syntax : (and <expr>...)

The logical and of a list of expressions. fsubr
<expr> the expressions to be ANDed

Returns : NIL if any expression evaluates to NIL, otherwise the value of the last expression (evaluation of expressions stops after the first expression that evaluates to NIL)

Or

Syntax : (or <expr>...)

The logical or of a list of expressions. fsubr
<expr> the expressions to be ORed

Returns : NIL if all expressions evaluate to NIL, otherwise the value of the first non-NIL expression (evaluation of expressions stops after the first expression that does not evaluate to NIL)

If

Syntax : (if <texpr> <expr1> [<expr2>])

Evaluate expressions conditionally. fsubr

<texpr> the test expression
<expr1> the expression to be evaluated if texpr is non-NIL
<expr2> the expression to be evaluated if texpr is NIL

Returns : the value of the selected expression

When

Syntax : (when <texpr> <expr>...)

Evaluate only when a condition is true. fsubr

<texpr> the test expression
<expr> the expression(s) to be evaluted if texpr is non-NIL

Returns : the value of the last expression or NIL

Unless

Syntax : (unless <texpr> <expr>...)

Evaluate only when a condition is false. fsubr

<texpr> the test expression

<expr> the expression(s) to be evaluated if texpr is NIL

Returns : the value of the last expression or NIL

Case

Syntax : (case <expr> <case>...[(t <expr>)])

Select by case. fsubr
- <expr> the selection expression
- <case> pair consisting of:
  - (<value> <expr>...)
  - where:
    - <value> is a single expression or a list of expressions (unevaluated)
    - <expr> are expressions to execute if the case matches
  - (t <expr>) default case (no previous matching)

Returns : the value of the last expression of the matching case

*Let,let**

Syntax : (let (<binding>...) <expr>...)

Create local bindings. fsubr.

Syntax : (let* (<binding>...) <expr>...)

Let with sequential binding. fsubr

<binding> the variable bindings each of which is either:

1) a symbol (which is initialized to NIL)

2) a list whose car is a symbol and whose cadr is an initialization expression

<expr> the expressions to be evaluated

Returns : the value of the last expression

Flet,labels,macrolet

Syntax : (flet (<binding>...) <expr>...)

Create local functions. fsubr

Syntax : (labels (<binding>...) <expr>...)

Flet with recursive functions. fsubr

Syntax : (macrolet (<binding>...) <expr>...)

Create local macros. fsubr

<binding> the function bindings each of which is:
(<sym> <fargs> <expr>...)
where:
<sym> the function/macro name
<fargs> formal argument list (lambda list)
<expr> expressions constituting the body of the function/macro
<expr> the expressions to be evaluated

Returns : the value of the last expression

Catch

Syntax : (catch <sym> <expr>...)

Evaluate expressions and catch throws. fsubr

<sym> the catch tag
<expr> expressions to evaluate

Returns : the value of the last expression the throw expression

Throw

Syntax : (throw <sym> [<expr>])

Throw to a catch. fsubr

<sym> the catch tag
<expr> the value for the catch to return (defaults to NIL)

Returns : never returns

Unwind-protect

Syntax : (unwind-protect <expr> <cexpr>...)

Protect evaluation of an expression. fsubr
<expr> the expression to protect
<cexpr> the cleanup expressions

Returns : the value of the expression
Note: unwind-protect guarantees to execute the cleanup expressions even if a non-local exit terminates the evaluation of the protected expression

Looping constructs

loop
do
do*
dolist
dotimes

Loop

Syntax : (loop <expr>...)

Basic looping form. fsubr
<expr> the body of the loop

Returns : never returns (must use non-local exit, such as RETURN)

*Do,do**

Syntax : (do (<binding>...) (<texpr> <rexpr>...) <expr>...)

Syntax : (do* (<binding>...) (<texpr> <rexpr>...) <expr>...)

General looping form. fsubr. do binds simultaneously, do* binds sequentially

<binding> the variable bindings each of which is either:

1) a symbol (which is initialized to NIL)
2) a list of the form: (<sym> <init> [<step>])

where:

<sym> is the symbol to bind
<init> the initial value of the symbol
<step> a step expression

<texpr> the termination test expression
<rexpr> result expressions (the default is NIL)
<expr> the body of the loop (treated like an implicit prog)

Returns : the value of the last result expression

Dolist

Syntax : (dolist (<sym> <expr> [<rexpr>]) <expr>...)

Loop through a list. fsubr

<sym> the symbol to bind to each list element
<expr> the list expression
<rexpr> the result expression (the default is NIL)
<expr> the body of the loop (treated like an implicit prog)

Returns : the result expression

Dotimes

Syntax : (dotimes (<sym> <expr> [<rexpr>]) <expr>...)

Loop from zero to N-1. fsubr

<sym> the symbol to bind to each value from 0 to n-1
<expr> the number of times to loop
<rexpr> the result expression (the default is NIL)
<expr> the body of the loop (treated like an implicit prog)

Returns : the result expression

The program feature

prog
prog*
block
return
return-from
tagbody
go
progv
prog1
prog2
progn

*Prog,prog**

Syntax : (prog <binding>...) <expr>...)

The program feature

Syntax : (prog* (<binding>...) <expr>...)

Prog with sequential binding. fsubr -- equivalent to (let () (block NIL (tagbody ...)))

<binding> the variable bindings each of which is either:

1) a symbol (which is initialized to NIL)
2) a list whose car is a symbol and whose cadr is an initialization expression

<expr> expressions to evaluate or tags (symbols)

Returns : NIL or the argument passed to the return function

Block

Syntax : (block <name> <expr>...)

Named block. fsubr
<name> the block name (quoted symbol)
<expr> the block body

Returns : the value of the last expression

Return

Syntax : (return [<expr>])

Cause a prog construct to return a value. fsubr
<expr> the value (defaults to NIL)

Returns : never returns

Return-from

Syntax : (return-from <name> [<value>])

Return from a named block or function. fsubr. In xlisp, the names are dynamically scoped.

<name> the block or function name (quoted symbol). If name is NIL, use function RETURN.

<value>the value to return (defaults to NIL)

Returns : never returns

Tagbody

Symbols : (tagbody <expr>...)

Block with labels. fsubr
<expr> expression(s) to evaluate or tags (symbols)

Returns : NIL

Go

Syntax : (go <sym>)

Go to a tag within a TAGBODY . fsubr. In xlisp, tags are dynamically scoped.
<sym> the tag (quoted)

Returns : never returns

Progv

Syntax : (progv <slist> <vlist> <expr>...)

Dynamically bind symbols. fsubr

<slist> list of symbols (evaluated)
<vlist> list of values to bind to the symbols (evaluated)
<expr> expression(s) to evaluate

Returns : the value of the last expression

Prog1

Syntax : (prog1 <expr1> <expr>...)

Execute expressions sequentially. fsubr

<expr1> the first expression to evaluate
<expr> the remaining expressions to evaluate

Returns : the value of the first expression

Prog2

Syntax : (prog2 <expr1> <expr2> <expr>...)

Execute expressions sequentially. fsubr

<expr1>	the first expression to evaluate
<expr2>	the second expression to evaluate
<expr>	the remaining expressions to evaluate

Returns : the value of the second expression

Progn

Syntax : (progn <expr>...)

Execute expressions sequentially. fsubr
<expr> the expressions to evaluate

Returns : the value of the last expression (or NIL)

Input/output functions

Note that when printing objects, printing is accomplished by sending the message :prin1 to the object.

read
set-macro-character
get-macro-character
print
prin1
princ
pprint
terpri
fresh-line
flatsize
flatc
y-or-n-p

Read

Syntax : (read [<stream> [<eof> [<rflag>]]])

Read an expression

<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)
<eof> the value to return on end of file (default is NIL)
<rflag> recursive read flag. The value is ignored

Returns : the expression read

Set-macro-character

Syntax : (set-macro-character <ch> <fcn> [T])

Modify read table. Defined in init.lsp

<ch>	character to define
<fcn>	function to bind to character.
T	if TMACRO rather than NMACRO

Get-macro-character

Syntax : (get-macro-character <ch>)

Examine read table. Defined in init.lsp

<ch>	character

Returns : function bound to character

Print

Syntax : (print <expr> [<stream>])

Print an expression on a new line. The expression is printed using prin1, then current line is terminated (Note: this is backwards from Common Lisp).

<expr> the expression to be printed
<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : the expression

Prin1

Syntax : (prin1 <expr> [<stream>])

Print an expression. Symbols, cons cells (without circularities), arrays, strings, numbers, and characters are printed in a format generally acceptable to the read function. Printing format can be affected by the global formatting variables: *print-level* and *print-length* for lists and arrays, *integer-format* for fixnums, *float-format* for flonums, *ratio-format* for ratios, and *print-case* and *readtable-case* for symbols.

<expr> the expression to be printed
<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : the expression

Princ

Syntax : (princ <expr> [<stream>])

Print an expression without quoting. Like PRIN1 except symbols (including uninterned), strings, and characters are printed without using any quoting mechanisms.

<expr> the expressions to be printed
<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : the expression

Pprint

Syntax : (pprint <expr> [<stream>])

Pretty print an expression. Uses prin1 for printing.
<expr> the expressions to be printed
<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : the expression

Terpri

Syntax : (terpri [<stream>])

Terminate the current print line
<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : NIL

Fresh-line

Syntax : (fresh-line [<stream>])

Start a new line.

<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : t if a new line was started, NIL if already at the start of a line.

Flatsize

Syntax : (flatsize <expr>)

Length of printed representation using prin1
<expr> the expression

Returns : the length

Flatc

Syntax : (flatc <expr>)

Length of printed representation using princ
<expr> the expression

Returns : the length

Y-or-n-p

Syntax : (y-or-n-p [<fmt> [<arg>...]])

Ask a yes or no question. Defined in common.lsp. Uses *terminal-io* stream for interaction.

<fmt>	optional format string for question (see format function)
<arg>	arguments, if any, for format string

Returns : T for yes, NIL for no.

The FORMAT function

Syntax : (format <stream> <fmt> [<arg>...])

Do formatted output

<stream>	the output stream (T is *standard-output*)
<fmt>	the format string
<arg>	the format arguments

Returns : output string if <stream> is NIL, NIL otherwise

The format string can contain characters that should be copied directly to the output and formatting directives. The formatting directives are:

~A or ~a print next argument using princ
~S or ~s print next argument using prin1
~D or ~d print next argument integer
~E or ~e print next argument in exponential form
~F or ~f print next argument in fixed point form
~G or ~g print next argument using either ~E or ~F depending on magnitude
~% start a new line
~& start a new line if not on a new line
~t or ~T go to a specified column
~~ print a tilde character
~\n ignore return and following whitespace

The format directives can contain optional prefix and optional colon (:) or at-sign (@) modifiers between the tilde and directive character. Prefix characters are unsigned integers, or the character 'v' to indicate the number is taken from the next argument, or a single quote (') followed by a single character for those parameters that should be a single character.

For ~A and ~S the full form is:

~mincol,colinc,minpad,padchar:@A (or S)

If : is given, NIL will print as "()" rather than "NIL". The string is padded on the right (or left, if @ is given) with at least "minpad" copies of the "padchar". Padding characters are then inserted "colinc" characters at a time until the total width is at least "mincol". The defaults are 0 for mincol and minpad, 1 for colinc, and #\space for padchar. For example:

~15,,2,'.@A

The output is padded on the left with at least 2 periods until the output is at least 15 characters wide.

For ~D the full form is:

~mincol,padchar@D

If the argument is not a FIXNUM, then the format "~mincolA" is used. If "mincol" is specified then the number is padded on the left to be at least that many characters long using "padchar". "padchar" defaults to #\space. If @ is used and the value is positive, then a leading plus sign is printed before the first digit.

For ~E ~F and ~G the full form is:

~mincol,round,padchar@E (or F or G)

(This implementation is not Common Lisp compatible.) If the argument is not a real number (FIXNUM, RATIO, or FLONUM), then the format "~mincol,padcharD" is used. The number is printed using the C language e, f, or g formats. If the number could potentially take more than 100 digits to print, then F format is forced to E format, although some C libraries will do this at a lower number of digits. If "round" is specified, than that is the number of digits to the right of the decimal point that will be printed, otherwise six digits (or whatever is necessary in G format) are printed. In G format, trailing zeroes are deleted and exponential notation is used if the exponent of the number is greater than the precision or less than -4. If the @ modifier is used, a leading plus sign is printed before positive values. If "mincol" is specified, the number is padded on the left to be at least "mincol" characters long using "padchar". "padchar" defaults to #\space.

For ~% and ~~, the full form is ~n% or ~n~. "n" copies (default=1) of the character are output.

For ~&, the full form is ~n&. ~0& does nothing. Otherwise enough new line characters are emited to move down to the "n"th new line (default=1).

For ~T, the full form is:

~count,tabwidth@T

The cursor is moved to column "count" (default 1). If the cursor is initially at count or beyond, then the cursor is moved forward to the next position that is a multiple of "tabwidth" (default 1) columns beyond count. When the @ modifier is used, then positioning is relative. "count" spaces are printed, then additional spaces are printed to make the column number be a multiple of "tabwidth". Note that column calcuations will be incorrect if ASCII tab characters or ANSI cursor positioning sequences are used.

For ~\n, if the colon modifier is used, then the format directive is ignored (allowing embedded returns in the source for enhanced readability). If the at-sign modifier is used, then a carriage return is emitted, and following whitespace is ignored.

File I/O functions

Note that initially, when starting XLISP-PLUS, there are six system stream symbols which are associated with three streams. *TERMINAL-IO* is a special stream that is bound to the keyboard and display, and allows for interactive editing. *STANDARD-INPUT* is bound to standard input or to *TERMINAL-IO* if not redirected. *STANDARD-OUTPUT* is bound to standard output or to *TERMINAL-IO* if not redirected. *ERROR-OUTPUT* (error message output), *TRACE-OUTPUT* (for TRACE and TIME functions), and *DEBUG-IO* (break loop i/o, and messages) are all bound to *TERMINAL-IO*. Standard input and output can be redirected on most systems.

File streams are printed using the #< format that cannot be read by the reader. Console, standard input, standard output, and closed streams are explicitly indicated. Other file streams will typically indicate the name of the attached file.

When the transcript is active (DRIBBLE function) , all characters that would be sent to the display via *TERMINAL-IO* are also placed in the transcript file.

TERMINAL-IO should not be changed. Any other system streams that are changed by an application should be restored to their original values.

read-char
peek-char
write-char
read-line
open
close
delete-file
truename
with-open-file
read-byte
write-byte
file-length
file-position

See also : File I/O examples

Read-char

Syntax : (read-char [<stream>])

Read a character from a stream
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)

Returns : the character or NIL at end of file

Peek-char

Syntax : (peek-char [<flag> [<stream>]])

Peek at the next character.

<flag> flag for skipping white space (default is NIL)
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)

Returns : the character or NIL at end of file

Write-char

Syntax : (write-char <ch> [<stream>])

Write a character to a stream

<ch>	the character to write
<stream>	the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : the character

Read-line

Syntax : (read-line [<stream>])

Read a line from a stream
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)

Returns : the string excluding the #\newline, or NIL at end of file

Open

Syntax : (open <fname> &key :direction :element-type :if-exists :if-does-not-exist)

Open a file stream. The function OPEN has been significantly enhanced over original XLISP. The original function only had the :direction keyword argument, which could only have the values :input or :output. When used with the :output keyword, it was equivalent to (open <fname> :direction :output :if-exists :supersede). A maximum of ten files can be open at any one time, including any files open via the LOAD, DRIBBLE, SAVE and RESTORE commands. The open command may force a garbage collection to reclaim file slots used by unbound file streams.

<fname>	the file name string, symbol, or file stream created via OPEN. In the last case, the name is used to open a second stream on the same file -- this can cause problems if one or more streams is used for writing.
:direction	Read and write permission for stream (default is :input).
:input	Open file for read operations only.
:probe	Open file for reading, then close it (use to test for file existance)
:output	Open file for write operations only.
:io	Like :output, but reading also allowed.
:element-type	FIXNUM or CHARACTER (default is CHARACTER), as returned by type-of function. Files opened with type FIXNUM are binary files instead of ascii, which means no crlf to/from lf conversion takes place, and control-Z will not terminate an input file. It is the intent of Common Lisp that binary files only be accessed with read-byte and write-byte while ascii files be accessed with any function but read-byte and write-byte. XLISP does not enforce that distinction.
:if-exists	action to take if file exists. Argument ignored for :input (file is positioned at start) or :probe (file is closed)
:error	give error message
:rename	rename file to generated backup name, then open a new file of the original name. This is the default action
:new-version	same as :rename
:overwrite	file is positioned to start, original data intact
:append	file is positioned to end
:supersede	delete original file and open new file of the same name
:rename-and-delete	same as :supersede
NIL	close file and return NIL
:if-does-not-exist	action to take if file does not exist.
:error	give error message (default for :input, or :overwrite or :append)
:create	create a new file (default for :output or :io when not :overwrite or :append)
NIL	return NIL (default for :probe)

Returns : a file stream, or sometimes NIL

Close

Syntax : (close <stream>)

Close a file stream. The stream becomes a "closed stream." Note that unbound file streams are closed automatically during a garbage collection. <stream> the stream, which may be a string stream

Returns : t if stream closed, NIL if terminal (cannot be closed) or already closed.

Delete-file

Syntax : (delete-file <fname>)

Delete a file.

<fname> file name string, symbol or a stream opened with OPEN

Returns : t if file does not exist or is deleted. If <fname> is a stream, the stream is closed before the file is deleted. An error occurs if the file cannot be deleted.

Truename

Syntax : (truename <fname>)

Obtain the file path name
<fname> file name string, symbol, or a stream opened with OPEN

Returns : string representing the true file name (absolute path to file).

With-open-file

Syntax : (with-open-file (<var> <fname> [<karg>...]) [<expr>...])

Evaluate using a file. Defined in common.lsp as a macro. File will always be closed upon completion

<var>	symbol name to bind stream to while evaluating expresssions (quoted)
<fname>	file name string or symbol
<karg>	keyword arguments for the implicit open command
<expr>	expressions to evaluate while file is open (implicit progn)

Returns : value of last <expr>.

Read-byte

Syntax : (read-byte [<stream>])

Read a byte from a stream.
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)

Returns : the byte (integer) or NIL at end of file

Write-byte

Syntax : (write-byte <byte> [<stream>])

Write a byte to a stream
<byte> the byte to write (integer)
<stream> the output stream (default, or NIL, is *standard-output*, T is *terminal-io*)

Returns : the byte (integer)

File-length

Syntax : (file-length <stream>)

Get length of file. For ascii file, the length reported may be larger than the number of characters read or written because of CR conversion

<stream> the file stream (should be disk file)

Returns : length of file, or NIL if cannot be determined.

File-position

Syntax : (file-position <stream> [<expr>])

Get or set file position. For an ascii file, the file position may not be the same as the number of characters read or written because of CR conversion. It will be correct when using file-position to position a file at a location earlier reported by file-position.

<stream>	the file stream (should be a disk file)
<expr>	desired file position, if setting position. Can also be :start for start of file or :end for end of file.

Returns : if setting position, and successful, then T; if getting position and successful then the position; otherwise NIL

String stream functions

These functions operate on unnamed streams. An unnamed output stream collects characters sent to it when it is used as the destination of any output function. The functions get-output-stream return a string or list of the characters.

An unnamed input stream is setup with the make-string-input-stream function and returns each character of the string when it is used as the source of any input function.

Note that there is no difference between unnamed input and output streams. Unnamed input streams may be written to by output functions, in which case the characters are appended to the tail end of the stream. Unnamed output streams may also be (destructively) read by any input function as well as the get-output-stream functions.

make-string-input-stream
make-string-output-stream
get-output-stream-string
get-output-stream-list
with-input-from-string
with-output-to-string

Make-string-input-stream

Syntax : (make-string-input-stream <str> [<start> [<end>]])
<str> the string
<start> the starting offset
<end> the ending offset + 1 or NIL for end of string

Returns : an unnamed stream that reads from the string

Make-string-output-stream

Syntax : (make-string-output-stream)

Returns : an unnamed output stream

Get-output-stream-string

Syntax : (get-output-stream-string <stream>)

The output stream is emptied by this function
<stream> the output stream

Returns : the output so far as a string

Get-output-stream-list

Syntax : (get-output-stream-list <stream>)

The output stream is emptied by this function
<stream> the output stream

Returns : the output so far as a list

With-input-from-string

Syntax : (with-input-from-string (<var> <str> &key :start :end :index) [<expr>...])

Defined in common.lsp as a macro

<var>	symbol that stream is bound to during execution of expressions (quoted)
<str>	the string
:start	starting offset into string (default 0)
:end	ending offset + 1 (default, or NIL, is end of string)
:index	setf place form which gets final index into string after last expression is executed (quoted)
<expr>	expressions to evaluate (implicit progn)

Returns : the value of the last <expr>

With-output-to-string

Syntax : (with-output-to-string (<var>) [<expr>...])

Defined in common.lsp as a macro

<var> symbol that stream is bound to during execution of expressions (quoted)
<expr> expressions to evaluate (implicit progn)

Returns : contents of stream, as a string

Debugging and error handling functions

trace
untrace
error
cerror
clean-up
top-level
errset
baktrace
evalhook
applyhook
debug
nodebug

Trace

Syntax : (trace [<sym>...])

Add a function to the trace list. fsubr
<sym> the function(s) to add (quoted)

Returns : the trace list

Untrace

Syntax : (untrace [<sym>...])

Remove a function from the trace list fsubr. If no functions given, all functions are removed from the trace list.

<sym> the function(s) to remove (quoted)

Returns : the trace list

Error

Syntax : (error <emsg> [<arg>])

Signal a non-correctable error

<emsg>	the error message string
<arg>	the argument expression (printed after the message)

Returns : never returns

Cerror

Syntax : (cerror <cmsg> <emsg> [<arg>])

Signal a correctable error

<cmsg>the continue message string
<emsg> the error message string
<arg> the argument expression (printed after the message)

Returns : NIL when continued from the break loop

Break

Syntax : (break [<bmsg> [<arg>]])

Enter a break loop

<bmsg>	the break message string (defaults to "**BREAK**")
<arg>	the argument expression (printed after the message)

Returns : NIL when continued from the break loop

Clean-up

Syntax : (clean-up)

Clean up after an error

Returns : never returns

Top-level

Syntax : (top-level)

Clean-up after an error and return to the top level

Returns : never returns

Continue

Syntax : (continue)

Continue from a correctable error

Returns : never returns

Errset

Syntax : (errset <expr> [<pflag>])

Trap errors. fsubr

<expr> the expression to execute
<pflag> flag to control printing of the error message (default t)

Returns : the value of the last expression consed with NIL or NIL on error

Baktrace

Syntax : (baktrace [<n>])

Print N levels of trace back information

<n>	the number of levels (defaults to all levels)

Returns : NIL

Evalhook

Syntax : (evalhook <expr> <ehook> <ahook> [<env>])

Evaluate with hooks

<expr> the expression to evaluate. <ehook> is not used at the top level.

<ehook> the value for *evalhook*

<ahook> the value for *applyhook*

<env> the environment (default is NIL). The format is a dotted pair of value (car) and function (cdr) binding lists. Each binding list is a list of level binding a-lists, with the innermost a-list first. The level binding a-list associates the bound symbol with its value.

Returns : the result of evaluating the expression

Applyhook

Syntax : (applyhook <fun> <arglist> <ehook> <ahook>)

Apply with hooks

<fun>	The function closure. <ahook> is not used for this function application.
<arglist>	The list of arguments.
<ehook>	the value for *evalhook*
<ahook>	the value for *applyhook*

Returns : the result of applying <fun> to <arglist>

Debug,nodebug

Syntax : (debug)

Enable debug breaks

Syntax : (nodebug)

Disable debug breaks

Defined in init.lsp

System functions

load
restore
save
savefun
dribble
gc
expand
alloc
room
time
get-internal-real-time
get-internal-run-time
coerce
type-of
peek
poke
address-of
get-key
system
exit
generic

Load

Syntax : (load <fname> &key :verbose :print)

Load a source file. An implicit ERRSET exists in this function so that if error occurs during loading, and *breakenable* is NIL, then the error message will be printed and NIL will be returned. The OS environmental variable XLPATH is used as a search path for files in this function. If the filename does not contain path separators ('/' for UNIX, and either '/' or '\' for MS-DOS) and XLPATH is defined, then each pathname in XLPATH is tried in turn until a matching file is found. If no file is found, then one last attempt is made in the current directory. The pathnames are separated by either a space or semicolon, and a trailing path separator character is optional.

<fname>	the filename string, symbol, or a file stream created with OPEN. The extension "lsp" is assumed.
:verbose	the verbose flag (default is t)
:print	the print flag (default is NIL)

Returns : t if successful, else NIL

Restore

Syntax : (restore <fname>)

Restore workspace from a file. The OS environmental variable XLPATH is used as a search path for files in this function. See the note under function load, above. The standard system streams are restored to the defaults as of when XLISP-PLUS was started. Files streams are restored in the same mode they were created, if possible, and are positioned where they were at the time of the save. If the files have been altered or moved since the time of the save, the restore will not be completely successful. Memory allocation will not be the same as the current settings of ALLOC are used. Execution proceeds at the top-level read-eval-print loop. The state of the transcript logging is not affected by this function.

<fname> the filename string, symbol, or a file stream created with OPEN. The extension "wks" is assumed.

Returns : NIL on failure, otherwise never returns

Save

Syntax : (save <fname>)

Save workspace to a file. You cannot save from within a load. Not all of the state may be saved -- see restore. By saving a workspace with the name "xlisp", that workspace will be loaded automatically when you invoke XLISP-PLUS.

<fname> the filename string, symbol, or a file stream created with OPEN. The extension "wks" is assumed.

Returns : t if workspace was written, NIL otherwise

Savefun

Syntax : (savefun <fcn>)

Save function to a file. defined in init.lsp
<fcn> function name (saves it to file of same name, with extension ".lsp")

Returns : t if successful

Dribble

Syntax : (dribble [<fname>])

Create a file with a transcript of a session

<fname> file name string, symbol, or file stream created with OPEN (if missing, close current transcript)

Returns : t if the transcript is opened, NIL if it is closed

Gc

Syntax : (gc)

Force garbage collection.

Returns : NIL

Expand

Syntax : (expand [<num>])

Expand memory by adding segments
<num> the number of segments to add, default 1

Returns : the number of segments added

Alloc

Syntax : (alloc <num> [<num2>])

Change segment size.

<num> the number of nodes to allocate

<num2> the number of pointer elements to allocate in an array segment (when dynamic array allocation compiled). Default is no change.

Returns : the old number of nodes to allocate

Room

Syntax : (room)

Show memory allocation statistics. Statistics (which are sent to *STANDARD-OUTPUT*) include:

Nodes - number of nodes, free and used
Free nodes - number of free nodes
Segments - number of node segments, including those reserved for characters and small integers.
Allocate - number of nodes to allocate in any new node segments
Total - total memory bytes allocated for node segments, arrays , and strings
Collections - number of garbage collections

When dynamic array allocation is compiled, the following additional statistics are printed:

Vector nodes - number of pointers in arrays and (size equivalent) strings
Vector segs - number of vector segments. Increases and decreases as needed.
Vec allocate - number of pointer elements to allocate in any new vector segment

Returns : NIL

Time

Syntax : (time <expr>)

Measure execution time. fsubr.
<expr> the expression to evaluate

Returns : the result of the expression. The execution time is printed to *TRACE-OUTPUT*

Get-internal-real-time,get-internal-run-time

Syntax : (get-internal-real-time)

Get elapsed clock time.

Syntax : (get-internal-run-time)

Get elapsed execution time.

Returns : integer time in system units (see internal-time-units-per-second) meaning of absolute values is system dependent.

Coerce

Syntax : (coerce <expr> <type>)

Force expression to designated type. Sequences can be coerced into other sequences, single character strings or symbols with single character printnames can be coerced into characters, fixnums can be coerced into characters or flonums. Ratios can be coerced into flonums. Flonums and ratios can be coerced into complex (so can fixnums, but they turn back into fixnums).

<expr> the expression to coerce
<type> desired type, as returned by type-of

Returns : <expr> if type is correct, or converted object.

Type-of

Syntax : (type-of <expr>)

Returns the type of the expression. It is recommended that typep be used instead, as it is more general. In the original XLISP, the value NIL was returned for NIL.

<expr> the expression to return the type of

Returns : One of the symbols:

LIST	for NIL (lists, conses return CONS)
SYMBOL	for symbols
OBJECT	for objects
CONS	for conses
SUBR	for built-in functions
FSUBR	for special forms
CLOSURE	for defined functions
STRING	for strings
FIXNUM	for integers
RATIO	for ratios
FLONUM	for floating point numbers
COMPLEX	for complex numbers
CHARACTER	for characters
FILE-STREAM	for file pointers
UNNAMED-STREAM	for unnamed streams
ARRAY	for arrays
HASH-TABLE	for hash tables
sym	for structures of type "sym"

Peek

Syntax : (peek <addrs>)

Peek at a location in memory.

<addrs>	the address to peek at (integer)

Returns : the value at the specified address (integer)

Poke

Syntax : (poke <addrs> <value>)

Poke a value into memory.
<addrs> the address to poke (integer)
<value>the value to poke into the address (integer)

Returns : the value

Address-of

Syntax : (address-of <expr>)

Get the address of an XLisp node
    <expr> the node

Returns : the address of the node (integer)

Get-key

Syntax : (get-key)

Read a keystroke from console. OS dependent.

Returns : integer value of key (no echo)

System

Syntax : (system <command>)

Execute a system command. OS dependent -- not always available.
<command> Command string, if 0 length then spawn OS shell

Returns : T if successful (note that MS/DOS command.com always returns success)

Exit

Syntax : (exit)

Exit XLisp

Returns : never returns

Generic

Syntax : (generic <expr>)

Create a generic typed copy of the expression. Note: added function, Tom Almy's creation for debugging xlisp.

<expr> the expression to copy

Returns : NIL if value is NIL and NILSYMBOL compilation option not declared, otherwise if type is:

SYMBOL	copy as an ARRAY
OBJECT	copy as an ARRAY
CONS	(CONS (CAR <expr>)(CDR <expr>))
CLOSURE	copy as an ARRAY
STRING	copy of the string
FIXNUM	value
FLONUM	value
RATIO	value
CHARACTER	value
UNNAMED-STREAM	copy as a CONS
ARRAY	copy of the array
COMPLEX	copy as an ARRAY
HASH-TABLE	copy as an ARRAY
structure	copy as an ARRAY

Graphic functions

The following graphic and display functions represent an extension by Tom Almy. Although these functions are available under the Windows version of XLisp , they should not be used because the language does not support Windows message handling so an XLisp program cannot handle the WM_PAINT message appropriately. It means that everything drawn by these functions will disappear when resizing the XLisp window. Accessing the Windows resources is possible through an other program written in any accustomed high-level language (C,Pascal) which can use the XLisp as a server through the communication DLL.

cls
cleol
goto-xy
color
move
moverel
draw
drawrel

Cls

Syntax : (cls)

Clear display. Clear the display and position cursor at upper left corner.

Returns : nil

Cleol

Syntax : (cleol)

Clear to end of line. Clears current line to end.

Returns : nil

Goto-xy

Syntax : (goto-xy [<column> <row>])

Get or set cursor position. Cursor is repositioned if optional arguments are specified. Coordinates are clipped to actual size of display.

<column> 0-based column (x coordinate)
<row> 0-based row (y coordinate)

Returns : list of original column and row positions

Color

Syntax : (color <value>)

Set drawing color.
<value>Drawing color (not checked for validity)

Returns : <value>

Move,moverel

Syntax : (move <x1> <y1> [<x2> <y2> ...])

Absolute move

Syntax : (moverel <x1> <y2> [<x2> <y2> ...])

Relative move. For moverel, all coordinates are relative to the preceeding point.

<x1> <y1>	Moves to point x1,y1 in anticipation of draw.
<x2> <y2>	Draws to points specified in additional arguments.

Returns : T if succeeds, else NIL

Draw,drawrel

Syntax : (draw [<x1> <y1> ...])

Absolute draw

Syntax : (drawrel [<x1> <y1> ...])

Relative draw. For drawrel, all coordinates are relative to the preceeding point.
<x1> <y1> Point(s) drawn to, in order.

Returns : T if succeeds, else NIL

Communication DLL

The XLisp communication library resides in the XServer.DLL file and invoked automatically when the program is started. Through this library an other program can start the XLisp as a server , have XLisp commands executed by it , and terminate it.

All this program has to do is the following :

1. Includes the xserver.h header for the prototypes of the library functions.
2. Imports the functions from the DLL in the .DEF file like the following :

```
IMPORTS     XServer.XDStartServer
            XServer.XDTerminateServer
                  etc.
```

3. Processes the XL_REQ message.

The program calls the imported function in the following order

1. First calls the XDStartServer to launch XLisp in server mode. XLisp brings up as an icon. If more client tasks want to use the server , only the first XDStartServer will actually load the program , the following start requests will be administrated by the communication DLL.
2. The client issues XDSendRequest call to send XLisp command to the server. The com library schedules the requests from different clients and passes them to the server.
3. When the server is ready with the processing , it notifies the client by sending XL_REQ message to its specified window. The window function of the client will respond by calling XDGetReply then XDDeleteReply. Now the client has the reply. The reply cannot be longer than 4KB.
4. Finishing its job the client terminates the server by the XDTerminateServer call. This action will terminate the server only if no more client tasks are logged to it.

XDStartServer

int FAR PASCAL XDStartServer(HWND Window);

This function starts the XLisp server if it has not been started yet by an other client.

Window	Handle to the main window of the client

Returns : 0 if succesful

XDSendRequest

int FAR PASCAL XDSendRequest(HWND Window , LPSTR Request);

Sends a request to the server. The caller will get back the control immediately after calling the function , the server starts to process the message when the client is descheduled after a GetMessage or PeekMessage call.

Window	Handle to the window to which the XL_REQ notification message is sent.
Request	A null-terminated string which will be passed to XLisp as command line. The line must be closed by CR-LF else XLisp will not start the evaluation.

Returns : 0 if succesful

XDGetReply

LPSTR FAR PASCAL XDGetReply();

The client can call this function after receiving an XL_REQ message. XDGetReply will return a far pointer to a null-terminated string containing the reply of the server. You must copy this reply to a safe area immediately after calling this function BEFORE you call any Windows API function. The reply is not removed from the reply queue , you must call XDDeleteReply after you processed it.

Returns : Pointer to the server's reply

XDDeleteReply

int FAR PASCAL XDDeleteReply();

The client calls this function after processing the reply with XDGetReply. Calling this function the client notifies the com library that it can remove the posted reply from the reply queue.

Returns : 0 , if succesful

XDTerminateServer

int FAR PASCAL XDTerminateServer();

The client calls this function if it is about to terminate and wants to shut down the server. The server will not terminate if it has unprocessed request packet or other clients are still logged to it.

Returns : 0 , if succesful

Additional functions and utilities

STEP.LSP
PP.LSP
REPAIR.LSP

Step.lsp

This file contains a simple Lisp single-step debugger. It started as an implementation of the "hook" example in chapter 20 of Steele's "Common Lisp". This version was brought up on Xlisp 1.7 for the Amiga, and then on VAXLISP.

To invoke: (step (whatever-form with args))

For each list (interpreted function call), the stepper prints the environment and the list, then enters a read-eval-print loop. At this point the available commands are:

(a list)<CR> evaluate the list in the current environment, print the result, and repeat.
<CR> step into the called function
anything_else<CR> step over the called function.

If the stepper comes to a form that is not a list it prints the form and the value, and continues on without stopping.

Note that stepper commands are executed in the current environment. Since this is the case, the stepper commands can change the current environment. For example, a SETF will change an environment variable and thus can alter the course of execution.

Global variables - newline, *hooklevel*

Functions/macros - while step eval-hool-function step-spaces step-flush

Note : an even more powerful stepper package is in stepper.lsp (documented in stepper.doc).

Pp.lsp

In addition to the pretty-printer itself, this file contains a few functions that illustrate some simple but useful applications.

Syntax : (pp <object> [<stream>])

Pretty print expression

Syntax : (pp-def <funct> [<stream>])

Pretty print function/macro

Syntax : (pp-file <file> [<stream>])

Pretty print file

<object>	The expression to print
<funct>	Function to print (as DEFUN or DEFMACRO)
<file>	File to print (specify either as string or quoted symbol)
<stream>	Output stream (default is *standard-output*)

Returns : T

Global variables: tabsize maxsize miser-size min-miser-car max-normal-car

Functions/Macros: sym-function pp-file pp-def make-def pp pp1 moveto spaces pp-rest-across pp-rest printmacrop pp-binding-form pp-do-form pp-defining-form pp-pair-form

See the source file for more information.

Repair.lsp

This file contains a structure editor.
Execute
(repair 'symbol) to edit a symbol.
(repairf symbol) to edit the function binding of a symbol (allows changing the argument list or function type, lambda or macro).

The editor alters the current selection by copying so that aborting all changes is generally posible; the exception is when editing a closure, if the closure is BACKed out of, the change is permanent.

For all commands taking a numeric argument, the first element of the selection is the 0th (as in NTH function).

Any array elements become lists when they are selected, and return to arrays upon RETURN or BACK commands.

Do not create new closures, because the environment will be incorrect. Closures become LAMBDA or MACRO expressions as the selection. Only the closure body may be changed; the argument list cannot be successfully modified, nor can the environment.

For class objects, only the methods and message names can be modified. For instance objects, instance variables can be examined (if the object under-stands the message :<ivar> for the particular ivar), and changed (if :SET-IVAR is defined for that class, as it is if CLASSES.LSP is used).

COMMANDS (general):
- ? list available commands for the selection.
- RETURN exit, saving all changes.
- ABORT exit, without changes.
- BACK go back one level (as before CAR CDR or N commands).
- B n go back n levels.
- L display selection using pprint; if selection is symbol, give short description.
- MAP pprints each element of selection, or if selection is symbol then gives complete description of properties.
- PLEV x set *print-level* to x. (Initial default is *rep-print-level*)
- PLEN x set *print-length to x. (Initial default is *rep-print-length*)
- EVAL x evaluates x and prints result. The symbol @ is bound to the selection.
- REPLACE x replaces the current selection with evaluated x. The symbol @ is bound to the selection.

COMMANDS (if selection is symbol):
- VALUE edit the value binding.
- FUNCTION edit the function binding (must be a closure).
- PROP x edit property x.

COMMANDS (if selection is list):
- CAR select the CAR of the current selection.
- CDR select the CDR of the current selection.
- n where n is small non-negative integer, changes current selection to (NTH n list).
- SUBST x y all occurances of (quoted) y are replaced with (quoted) x. EQUAL is used for the comparison.
- RAISE n removes parenthesis surrounding nth element of

selection.
LOWER n m inserts parenthesis starting with the nth element, for
m elements.
ARRAY n m as in LOWER, but makes elements into an array.
I n x inserts (quoted) x before nth element in selection.
R n x replaces nth element in selection with (quoted) x.
D n deletes nth element in selection.

All function names and global variables start with the string "rep-" or "*rep-*".

Examples : File I/O functions

Input from a File

To open a file for input, use the OPEN function with the keyword argument :DIRECTION set to :INPUT. To open a file for output, use the OPEN function with the keyword argument :DIRECTION set to :OUTPUT. The OPEN function takes a single required argument which is the name of the file to be opened. This name can be in the form of a string or a symbol. The OPEN function returns an object of type FILE-STREAM if it succeeds in opening the specified file. It returns the value NIL if it fails. In order to manipulate the file, it is necessary to save the value returned by the OPEN function. This is usually done by assigning it to a variable with the SETQ special form or by binding it using LET or LET*. Here is an example:

```
(setq fp (open "init.lsp" :direction :input))
```

Evaluating this expression will result in the file "init.lsp" being opened. The file object that will be returned by the OPEN function will be assigned to the variable "fp".

It is now possible to use the file for input. To read an expression from the file, just supply the value of the "fp" variable as the optional "stream" argument to READ.

```
(read fp)
```

Evaluating this expression will result in reading the first expression from the file "init.lsp". The expression will be returned as the result of the READ function. More expressions can be read from the file using further calls to the READ function. When there are no more expressions to read, the READ function will return NIL (or whatever value was supplied as the second argument to READ).

Once you are done reading from the file, you should close it. To close the file, use the following expression:

```
(close fp)
```

Evaluating this expression will cause the file to be closed.

Output to a File

Writing to a file is pretty much the same as reading from one. You need to open the file first. This time you should use the OPEN function to indicate that you will do output to the file. For example:

```
(setq fp (open "test.dat" :direction :output :if-exists :supersede))
```

Evaluating this expression will open the file "test.dat" for output. If the file already exists, its current contents will be discarded. If it doesn't already exist, it will be created. In any case, a FILE-STREAM object will be returned by the OPEN function. This file object will be assigned to the "fp" variable.

It is now possible to write to this file by supplying the value of the "fp" variable as the optional "stream" parameter in the PRINT function.

```
(print "Hello there" fp)
```

Evaluating this expression will result in the string "Hello there" being written to the file "test.dat". More data can be written to the file using the same technique.

Once you are done writing to the file, you should close it. Closing an output file is just like closing an input file.

(close fp)

Evaluating this expression will close the output file and make it permanent.

A Slightly More Complicated File Example

This example shows how to open a file, read each Lisp expression from the file and print it. It demonstrates the use of files and the use of the optional "stream" argument to the READ function.

```
(do* ((fp (open "test.dat" :direction :input))
      (ex (read fp) (read fp)))
     ((null ex) (close fp) nil)
  (print ex))
```

The file will be closed with the next garbage collection.