# XLISP-PLUS: Another Object-oriented Lisp

Version 3.0

January 15, 1997

Tom Almy
almy@teleport.com

Portions of this manual and software are from XLISP which is Copyright (c) 1988, by David Michael Betz, all rights reserved. Mr. Betz grants permission for unrestricted non-commercial use. Portions of XLISP-PLUS from XLISP-STAT are Copyright (c) 1988, Luke Tierney. UNIXSTUF.C is from Winterp 1.0, Copyright 1989 Hewlett-Packard Company (by Niels Mayer). Other enhancements and bug fixes are provided without restriction by Tom Almy, Mikael Pettersson, Neal Holtz, Johnny Greenblatt, Ken Whedbee, Blake McBride, Pete Yadlowsky, and Richard Zidlicky. See source code for details.

Table of Contents

# INTRODUCTION

XLISP-PLUS is an enhanced version of David Michael Betz's XLISP to have additional features of Comm on 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 XL ISP-PLUS with little modification. See the section starting on page 108 for details of the differences between XLISP and XLISP-PLUS.

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 i n this document.

XLISP-PLUS has a number of compilation options to to eliminate groups of functions and to tailor itself t o 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 almy@teleport.com, website http://www.teleport.com/~almy/xlisp.html. You can also reach Tom by writing to him at 17830 SW Shasta Trail, Tualatin, OR 97062, USA.

# 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 a path in XLPATH or the current directory. This file can be modified to suit the user's requirements. It contains a number of preference items.

If *startup-functions* is non-nil (default is nil), it is taken as a list of functions with no arguments which a re executed in sequence at this time. This allows automatically starting applications stored in workspaces.

If the variable *load-file-arguments* is non-nil (default is "t"), it then loads any files named as parameters on the command line (after appending ".lsp" to their names). If the "-v" flag is in the command line, then the files are loaded verbosely.

The option "-tfilename" will open a transcript file of the name "filename". At this time the top level comm and loop is entered. This is the function TOP-LEVEL-LOOP, by default.

XLISP then issues the following prompt (unless standard input has been redirected):

```
>
```

This indicates that XLISP is waiting for an expression to be typed. If the current package is other than US ER, the the package name is printed before the ">".

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 |

Under MS-DOS or OS/2 (at least) 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 |
| ctrl-T | print information |

Under MS-DOS if the global variable *dos-input* is set non-NIL, DOS is used to read entire input lines. Operation this way is convenient if certain DOS utilities, such as CED, are used, or if XLISP is run under an editor like EPSILON. In this case, normal command line editing is available, but the control keys will not work (in particular, ctrl-C will cause the program to exit!). Use the XLISP functions top-level, clean-up, a nd continue instead of ctrl-C, ctrl-G, and ctrl-P.

Under MS-DOS if the global variable *dos-input* is NIL, or under OS/2 or Windows, a special internal li ne editor is used. In this case the last 20 lines are saved, and can be recalled and viewed using the up and down arrow keys. Duplicate lines are not saved.

An additional feature is symbol name lookup. This command takes what appears to be an incomplete symbol name to the left of the cursor and prints all interned symbol names that match.

The control keys for the editor are:

| | |
|---|---|
| Up Arrow | Previous command in queue |
| Down Arrow | Next command in queue |
| Left Arrow | Move cursor to left |
| Right Arrow | Move cursor to right |
| Home | Move cursor to start of line |
| End | Move cursor to end of line |
| Delete | Delete character at cursor |
| Backspace | Delete character to left of cursor |
| Escape | Delete current line |
| Tab | Look up partial symbol name to left of cursor |

Characters are inserted at the current cursor position. Lines are limited in length to the width of the display, and invalid keystrokes cause the bell to ring.

## 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.

If XLISP was invoked with the command line argument "-b" then XLISP assumes it is running in batch mode. In batch mode any uncaught error will cause XLISP to exit after printing the error message.

# 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 and Windows implementations.

All data nodes are effectively cons cells consisting of two pointers 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
  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 (integers)
  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.
- bignums (integers)
  Big integers which don't fit in fixnums are stored in a special structure. Part of the bignum extension compilation option, when absent fixnums will overflow into flonums. Fixnums and flonums are collectively referred to as "integers". [size limit is about 65500 characters for printing or about 500000 bits for calculations].
- ratios
  The CAR field is used to hold the numerator while the CDR field is used to hold the denominator. The numerator and denominator are stored as either both bignums or both fixnums. All ratios results are returned in reduced form, and are returned as integers if the denominator is 1. Part of the bignums extension. Ratios and integers are collectively referred to as rationals.
- 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. Flonums and rational numbers are collectively referred to as real numbers.

- complex numbers
  Part of the math extension compilation option. The CAR field is used to hold the real part while the CDR field is used to hold the imaginary part. The parts can be either both rationals (ratio or integer) or both flonums. Any function which would return an integer complex number with a zero imaginary part returns just the real integer.
- 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 (file)
  The CAR and CDR fields are used in a system dependent way as a file pointer.
- streams (unnamed -- string)
  Implemented as a tconc-style list of characters.
- subrs (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 (special forms)
  Same implementation as subrs.
- closures (user defined functions)
  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 (see page 93 for format)
  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).
- packages
  Implemented using a structure. Packages must only be manipulated with the functions provided.

# 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 T is also reserved for use as the truth value. The symbol NIL represents an empty list.

Symbols starting with a colon are keywords, and will always evaluate to themselves. When the package facility is compiled as part of XLISP, colons have a special significance. Thus colons should not be used as part of a symbol name, except for these special uses.

Integer literals consist of a sequence of digits optionally beginning with a sign ('+' or '-'). Unless the bignum extension is used, 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. The radix of the literal is determined by the value of the variable *read-base* if it has an integer value within the range 2 to 36. However the literal can end with a period '.' in which case it is treated as a decimal number. It is generally not a good idea to assign a value to *read-base* unless you are reading from a file of integers in a non-decimal radix. Use the read macros instead to specify the base explicitly.

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 (not trailing) 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'. Conversely, symbols that could be interpreted as numeric literals in the current radix must have escape characters.

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 real number literal. If either field has a flonum literal, then both values are converted to flonums. Rational (integer or ratio) complex literals with a zero imaginary part are automatically reduced to rationals.

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 |

# 8 BIT ASCII CHARACTERS

When used in an IBM PC environment (or perhaps others), XLISP-PLUS is compiled by default to allow the full use of the IBM 8 bit ASCII character set, including all characters with diacritic marks. Note that using such characters will make programs non-portable. XLISP-PLUS can be compiled for standard 7 bit ASCII if desired for portability.

When 8 bit ASCII is enabled, the following system characteristics change:

Character codes 128 to 254 are marked as :constituent in the readtable. This means that any of the new characters (except for the nonprinting character 255) can be symbol constituent. Alphabetic characters which appear in both cases, such as é and É, are considered to be alphabetical for purposes of symbol case control, while characters such as á that have no coresponding upper case are not considered to be alphabetical.

The reader is extended for the character data type to allow all the additional characters (except code 255) to be entered literally, for instance "#\é". These characters are also printed literally, rather than using the "M-" construct. Code 255 must still be entered as, and will be printed as, "#\M-Rubout".

Likewise strings do not need and will not use the backslash escape mechanism for codes 128 to 254.

The functions alphanumericp, alpha-char-p, upper-case-p, and lower-case-p perform as would be expected on the extended characters, treating the diacritic characters as their unadorned counterparts. As per the Common Lisp definition, both-case-p will only indicate T for characters available in both cases.

## 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 i s 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 strea m and the second is the character that caused the invocation of the readmacro. The readmacro function sho uld return NIL to indicate that the character should be treated as white space or a value consed with NIL t o 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>
#d<digits>               == a decimal number (integer or ratio)
#x<hdigits>              == a hexadecimal integer or ratio (0-9,A-F)
#o<odigits>              == an octal integer or ratio (0-7)
#b<bdigits>              == a binary integer or ratio (0-1)
#<base>r<digits>         == an integer or ratio in base <base>, 2-36
#| |#                    == a comment
#:<symbol>               == an uninterned symbol
#C(r i)                  == a complex number
#+<expr>                 == conditional on feature expression true
#-<expr>                 == conditional on feature expression false
```

A feature expression is either a symbol or a list where the first element is AND, OR, or NOT and any remaining elements (NOT requires exactly one) are feature expressions. A symbol is true if it is a member (by test function EQ) of the list in global variable *FEATURES*. Init.lsp defines one initial feature, :XLISP, and the features :TIMES, :GENERIC, :POSFCNS (various position functions), :MATH (complex math), :BIGNUMS (bignums and ratios), :PC8 (character set), :PACKAGES, and :MULVALS depending on the coresponding feature having been compiled into the XLISP executable. Utility files supplied with XLISP-PLUS generally add new features which are EQ to the keyword made from their file names.

## 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 :DOWNC ASE :PRESERVE or :INVERT, while *PRINT-CASE* can have the values :UPCASE :DOWNCASE or :CAPITALIZE. 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 *PRINT-CASE* value of :CAPITALIZE means that the first character of the symbol, and any character in the symbol immediately following a non-alphabetical character are to be in uppercase, while all other alphabetical characters are to be in lowercase. The remaining *READTABLE-CASE* modes ignore *PRINT-CASE* and do not escape alphabetic characters. :PRESER VE never changes the case of characters while :INVERT inverts the case of any non mixed-case symbols.

There are five 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. Sym bols 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. Symbol s 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. Alph a quoting should be avoided. Predefined symbols print in lower case, other symbols print as they were entered.

**E: *READTABLE-CASE* :UPCASE *PRINT-CASE* :CAPITALIZE**

Like case B, except symbol names print capitalized.

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

# PACKAGES

When compiled in, XLISP-PLUS provides the "Packages" name hiding facility of Common Lisp. When in use, there are multiple object arrays (name spaces). Each package has internal and external symbols. Internal symbols can only normally be accessed while in that package, while external symbols can be imported into the current package and used as though they are members of the current package. There are three standard packages, XLISP, KEYWORD, and USER. In addition, some of the utility programs are in package TOOLS. Normally one is in package USER, which is initally empty. USER imports all external symbols from XLISP, which contains all the functions and variables described in the body of this document. Symbols which are not imported into the current package, but are declared to be external in their home package, can be referenced with the syntax "packageName:symbolName" to identify symbol *symbolName* in package *packageName*. Those symbols which are internal in their home package need the slightly more difficult syntax "packageName::symbolName".

The KEYWORD package is referenced by a symbol name with a leading colon. All keywords are in this package. All keywords are automatically marked external, and are interned as constants with themselves as their values.

To build an application in a package (to avoid name clashes, for instance), use MAKE-PACKAGE to create a new package (only if the package does not already exist, use FIND-PACKAGE to test first), and then preceed the application with the IN-PACKAGE command to set the package. Use the EXPORT function to indicate the symbols that will be accessable from outside the package.

To use an application in a package, either use IMPORT to make specific symbols accessable as local internal symbols, use USE-PACKAGE to make them all accessable, or explicitly reference the symbols with the colon syntax.

The file MAKEWKS.LSP shows how to build an initial XLISP workspace such that all the tools are accessable.

For the subtleties of the package facility, read Common Lisp the Language, second edition.

# 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. 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'.

If identical keywords occur, those after the first are ignored. Extra keywords will signal an error unless &allow-other-keys is present, in which case the extra keywords are ignored. Also, if the keyword :allow-other-keys is used in the function/macro call, and has a non-nil value, then additional keys will be 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 |

# GENERALIZED VARIABLES

Several XLISP functions support the concept of generalized variables. The idea behind generalized variables is that variables have two operations, access and update. Often two separate functions exist for the access and update operations, such as SYMBOL-VALUE and SET for the dynamic symbol value, or CAR and REPLACA for the car part of a cons cell. Code can be simplified if only one such function, the access function, is necessary. The function SETF is used to update. SETF takes a "place form" argument which specifies where the data is to be stored. The place form is identical to the function used for access. Thus we can use (setf (car x) 'y) instead of (rplaca x 'y). Updates using place forms are "destructive" in that the alter the data structure rather than rebuilding. Other functions which take place forms include PSETF, GETF, REMF, PUSH, PUSHNEW, POP, INCF, and DECF.

XLISP has a number of place forms pre-defined in code. In addition, the function DEFSETF can be used to define new place forms. Place forms and functions that take them as arguments must be carefully defined so that no expression is evaluated more than once, and evaluation proceeds from left to right. The result of this is that these functions tend to be slow. If multiple evaluation and execution order is not a concern, alternative simpler functions can be un-commented in COMMON.LSP and COMMON2.LSP.

A place form may be one of the following:

- A symbol (variable name). In this case note that (setf x y) is the same as (setq x y)
- A function call form of one of the following functions: CAR CDR NTH AREF ELT GET GETF SYMBOL-VALUE SYMBOL-FUNCTION SYMBOL-PLIST GETHASH. The function GETF itself has a place form which must aditionally be valid. The file COMMON2.LSP define additional placeforms (using DEFSETF) for LDB MASK-FIELD FIRST REST SECOND THIRD (through TENTH) and CxR. When used as a place form, the second argument of LDB and MASK-FIELD must be place forms and the number is not destructively modified.
- A macro form, which is expanded and re-evaluated as a place form.
- (send <obj> :<ivar>) to set the instance variable of an object (requires CLASSES.LSP be used).
- (<sym>-<element> <struct>) to set the element, <element>, of structure <struct> which is of type <sym>/
- (<fieldsym> <args>) form name <fieldsym>, defined with DEFSETF or manually, is used with the arguments. When used with SETF, the function stored in property *setf* of symbol <fieldsym> is applied to (<args> <setf expr>), or, alternatively, the function stored in property *setf-lambda* is applied then the result is evaluated in the current context.

# 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 HEIRARCHY

Messages:

:show — SHOW AN OBJECT'S INSTANCE VARIABLES
- returns: the object

:class — RETURN THE CLASS OF AN OBJECT
- returns: the class of the object

:prin1 [<stream>] — PRINT THE OBJECT
- <stream>: T is *terminal-io*, NIL does not print (for FLATSIZE calculation), and default is *standard-output*
- 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)

THE 'Class' CLASS

Class THE CLASS OF ALL OBJECT CLASSES (including itself)

Messages:

: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)

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.

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 (see page 20), 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. This variable does note exist when the package feature is compiled in.
- *package* - the current package. Do not alter. Part of the package feature.
- *terminal-io* - stream bound to keyboard and display. Do not alter.
- *standard-input* - the standard input stream, initially stdin. If stdin is not redirected on the command line, then *terminal-io* is used so that all interactive i/o uses the same stream.
- *standard-output* - the standard output stream, initially stdout. If stdout is not redirected on the command line then *terminal-io* is used so that all interactive i/o uses the same stream.
- *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 (see page 4)
- *tracelist* - list of names of functions to trace, as set by trace function.
- *tracenable* - enable trace back printout on errors (see page 4).
- *tracelimit* - number of levels of trace back information (see page 4).
- *evalhook* - user substitute for the evaluator function (see page 8, and evalhook and applyhook functions).
- *applyhook* - user substitute for function application (see page 8, and evalhook and applyhook functions).
- *readtable* - the current readtable (see page 12).
- *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 (see page 8).
- *integer-format* - format for printing integers (when not bound to a string, defaults to "%d" or "%ld" depending on implementation). Variable not used when bignum extension installed.
- *float-format* - format for printing floats (when not bound to a string, defaults to "%g")
- *readtable-case* - symbol read and output case. See page 14 for details
- *read-base* - When bound to a fixnum in the range 2 through 36, determines the default radix used when reading rational numbers. Part of bignum extension.
- *print-base* - When bound to a fixnum in the range 2 through 36, determines the radix used when printing rational numbers with prin1 and princ. Part of bignum extension.
- *print-case* - symbol output case when printing. See page 14 for details
- *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.
- *dos-input* - When not NIL, uses dos line input function for read (see page 3).
- *displace-macros* - When not NIL, macros are replaced by their expansions when exectuted (see page 7).
- *random-state* - the default random-state used by the random function.
- *features* - list of features, initially (:xlisp), used for #+ and #- reader macros.
- *startup-functions* - list of functions to be executed when workspace started
- *command-line* - the xlisp command line, in the form of a list of strings, one string per argument.
- *load-file-arguments* - When not NIL, file arguements are loaded at startup.
- *top-level-loop* - Top level loop to utilize, defaults to TOP-LEVEL-LOOP. Note that this function can only be restarted by executing TOP-LEVEL, and it never exits.
- *read-suppress* - When not NIL, inhibits certain parts of reading. Used by the #+ and #- macroes.

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 <expr>) EVALUATE AN XLISP EXPRESSION

<expr> the expression to be evaluated
returns the result of evaluating the expression

(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 <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 <expr>) RETURN AN EXPRESSION UNEVALUATED

fsubr
<expr> the expression to be quoted (quoted)
returns <expr> unevaluated

(function <expr>) GET THE FUNCTIONAL INTERPRETATION

fsubr
<expr> the symbol or lambda expression (quoted)
returns the functional interpretation

(complement <fun>) MAKE A COMPLEMENTARY FUNCTION

This function is intended to eliminate the need for -IF-NOT functions and :TEST-NOT keys by providing a way to make complementary functions.
<fun> the function or closure (not macro or fsubr)
returns a new function closure that returns NOT of the result of the original function.

(identity <expr>) RETURN THE EXPRESSION

<expr> the expression
returns the expression

(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 <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 <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 <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 <closure>) GET THE LAMBDA EXPRESSION

| | |
|---|---|
| <closure> | the closure |
| returns | the original lambda expression, or NIL if not a closure. Second return value is T if closure has a non-global environment, and the third return value is the name of the closure. |

(macroexpand <form>) RECURSIVELY EXPAND MACRO CALLS

| | |
|---|---|
| <form> | the form to expand |
| returns | the macro expansion |

(macroexpand-1 <form>) EXPAND A MACRO CALL

| | |
|---|---|
| <form> | the macro call form |
| returns | the macro expansion |

## MULTIPLE VALUE FUNCTIONS

XLISP-PLUS supports multiple return values (via a compilation option) as in Common Lisp. Note that most FSUBR control structure functions will pass back multiple return values, with the exceptions being PROG1 and PROG2.

(multiple-value-bind <varlist> <vform> [<form>...]) BIND RETURN VALUES INTO LOCAL CONTEXT

defined as macro in common.lsp

| | |
|---|---|
| <vform> | form to be evaluated |
| <varlist> | list of variables to bind to return values of vform |
| <form> | forms evaluated sequentially, as in LET, using local bindings |
| returns | values of last form evaluated, or NIL if no forms |

(multiple-value-call <fun> <form> ...) COLLECT VALUES AND APPLY FUNCTION

fsubr

| | |
|---|---|
| <fun> | function to apply |
| <form> | forms, which are evaluated, with result values collected |
| returns | result of applying fun to all of the returned values of the forms |

(multiple-value-list <form>) COLLECT MULTIPLE RETURNED VALUES INTO A LIST

defined as macro in common.lsp

| | |
|---|---|
| <form> | form to be evaluated |
| returns | list of returned values |

(multiple-value-prog1 <form> [<form> ...]) RETURN VALUES OF FIRST FORM

fsubr

| | |
|---|---|
| <form> | one or more forms, which are evaluated sequentially |
| returns | the result values of the first form |

(multiple-value-setq <varlist> <form>) BIND RETURN VALUES TO VARIABLES

defined as macro in common.lsp

| | |
|---|---|
| <form> | form to be evaluated |
| <varlist> | list of variables to bind to return values of form |
| returns | (undefined, implementation dependent) |

(nth-value <index> <form>) EXTRACT A RETURN VALUE

fsubr

| | |
|---|---|
| <index> | index into return values |
| <form> | form which gets evaluated |
| returns | the nth result value of exectuing the form |

(values [<expr>]) RETURN MULTIPLE VALUES

| | |
|---|---|
| <expr> | expression(s) to be evaluated |
| returns | each argument as a separate value |

(values-list <list>) RETURN MULTIPLE VALUES FROM LIST

defined in common.lsp

| | |
|---|---|
| <list> | a list |
| returns | each list element as a separate value |

## SYMBOL FUNCTIONS

(set <sym> <expr>) SET THE GLOBAL VALUE OF A SYMBOL

You can also use (setf (symbol-value <sym>) <expr>)

| | |
|---|---|
| <sym> | the symbol being set |
| <expr> | the new value |
| returns | the new value |

(setq [<sym> <expr>]...) SET THE VALUE OF A SYMBOL

fsubr. You can also use (setf <sym> <expr>)

| | |
|---|---|
| <sym> | the symbol being set (quoted) |
| <expr> | the new value |
| returns | the last new value or NIL if no arguments |

(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 | NIL |

(defun <sym> <fargs> <expr>...) DEFINE A FUNCTION
(defmacro <sym> <fargs> <expr>...) DEFINE A MACRO

fsubr

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

(gensym [<tag>]) GENERATE A SYMBOL

| | |
|---|---|
| <tag> | string or number |
| returns | the new symbol, uninterned |

(intern <pname> [<package>]) MAKE AN INTERNED SYMBOL

| | |
|---|---|
| <pname> | the symbol's print name string |
| <package> | the package (defaults to current package) |
| returns | the new symbol. A second value is returned which is NIL if the symbol did not pre-exist, :internal if it is an internal symbol, :external if it is an external symbol, or :inherited if it inherited via USE-PACKAGE. |

(make-symbol <pname>) MAKE AN UNINTERNED SYMBOL

| | |
|---|---|
| <pname> | the symbol's print name string |
| returns | the new symbol |

(symbol-name <sym>) GET THE PRINT NAME OF A SYMBOL

| | |
|---|---|
| <sym> | the symbol |
| returns | the symbol's print name |

(symbol-value <sym>) GET THE VALUE OF A SYMBOL

May be used as a place form.

| | |
|---|---|
| <sym> | the symbol |
| returns | the symbol's value |

(symbol-function <sym>) GET THE FUNCTIONAL VALUE OF A SYMBOL

May be used as a place form.

| | |
|---|---|
| <sym> | the symbol |
| returns | the symbol's functional value |

(symbol-plist <sym>) GET THE PROPERTY LIST OF A SYMBOL

May be used as a place form.

| | |
|---|---|
| <sym> | the symbol |
| returns | the symbol's property list |

(hash <expr> <n>) COMPUTE THE HASH INDEX

| | |
|---|---|
| <expr> | the object to hash |
| <n> | the table size (positive fixnum less than 32768) |
| returns | the hash index (fixnum 0 to n-1) |

(makunbound <sym>) MAKE A SYMBOL VALUE BE UNBOUND

You cannot unbind constants.

| | |
|---|---|
| <sym> | the symbol |
| returns | the symbol |

(fmakunbound <sym>) MAKE A SYMBOL FUNCTION BE UNBOUND

| | |
|---|---|
| <sym> | the symbol |
| returns | the symbol |

(unintern <sym> [<package>]) UNINTERN A SYMBOL

Defined in common.lsp if package extension not compiled.

| | |
|---|---|
| <sym> | the symbol |
| <package> | the package to look in for the symbol |
| returns | t if successful, NIL if symbol not interned |

(defconstant <sym> <val> [<comment>]) DEFINE A CONSTANT

fsubr.

| | |
|---|---|
| <sym> | the symbol |
| <val> | the value |
| <comment> | optional comment string (ignored) |
| returns | the value |

(defparameter <sym> <val> [<comment>]) DEFINE A PARAMETER

fsubr.

| | |
|---|---|
| <sym> | the symbol (will be marked "special") |
| <val> | the value |
| <comment> | optional comment string (ignored) |
| returns | the value |

(defvar <sym> [<val> [<comment>]]) DEFINE A VARIABLE

fsubr. Variable only initialized if not previously defined.

| | |
|---|---|
| <sym> | the symbol (will be marked "special") |
| <val> | the initial value, or NIL if absent. |
| <comment> | optional comment string (ignored) |
| returns | the current value |

(mark-as-special <sym> [<flag>]) SET SPECIAL ATTRIBUTE

Also see definition of PROCLAIM and DECLARE.

| | |
|---|---|
| <sym> | symbol to mark |
| <flag> | non-nil to make into a constant |
| returns | nil, with symbol marked as special and possibly as a constant. |

(declare [<declaration> ...]) DECLARE ARGUMENT ATTRIBUTES

Macro in common.lsp provided to assist in porting Common Lisp applications to XLISP-PLUS.

| | |
|---|---|
| <declaration> | list of local variable and attributes |
| returns | nil, produces an error message if attribute SPECIAL is used. |

(proclaim <proc>) PROCLAIM GLOBAL SYMBOL ATTRIBUTES

Function in common.lsp provided to assist in porting Common Lisp applications to XLISP-PLUS.

| | |
|---|---|
| <proc> | a list of symbols. If the CAR of the list is SPECIAL, then the remaining symbols are marked as special variables. |

(copy-symbol <sym> [<flag>]) MAKE A COPY OF A SYMBOL

Function in common2.lsp

| | |
|---|---|
| <sym> | symbol to copy |
| <flag> | if present and non-nil, copy value, function binding, and property list. |
| returns | un-interned copy of <sym> |

## GENERALIZED VARIABLE FUNCTIONS

(setf [<place> <expr>]...) SET THE VALUE OF A FIELD

fsubr

| | |
|---|---|
| <place> | the field specifier |
| <expr> | the new value |
| returns | the last new value, or NIL if no arguments |

(psetf [<place> <expr>]...) PARALLEL VERSION OF SETF

fsubr. All expressions are evaluated and macro place forms expanded before any assignments are made.

| | |
|---|---|
| <place> | the field specifier |
| <expr> | the new value |
| returns | NIL |

(defsetf <sym> <fcn>) DEFINE A SETF FIELD SPECIFIER
(defsetf <sym> <fargs> (<value>) <expr>...)

Defined as macro in common.lsp. Convenient, Common Lisp compatible alternative to setting *setf* or *setf-lambda* property directly.

| | |
|---|---|
| <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 of unevaluated arguments (lambda list) (quoted) |
| <value> | symbol bound to value to store (quoted). |
| <expr> | The last expression must an expression to evaluate in the setf context.In this respect, defsetf works like a macro definition. |
| returns | the field specifier symbol |

(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 <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 <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 <place> [<value>]) INCREMENT A FIELD
(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>) |

# PACKAGE FUNCTIONS

These functions are defined when the packages extension is compiled. The <package> argument can be either a string, symbol, or package object. The default when no package is given is the current package (as bound to *package*), unless otherwise specified in the definition. The <symbols> argument may be either a single symbol or a list of symbols. In case of name conflicts, a correctable error occurs.

When the packages extension is not compiled, simplified versions of apropos, apropos-list, and do-all-symbols are provided in common2.lsp. In addition, init.lsp will define dummy versions of export and in-package.

(apropos <string> [<package>]) SEARCH SYMBOLS FOR NAME MATCH
(apropos-list <string> [<package>])

Functions in common.lsp.

<string> find symbols which contain this string as substring of print name
<package> package to search, if absent, or NIL, search all packages
returns apropos-list returns list of symbols, apropos prints them, along with some information, and returns nothing.

(defpackage <package> [<option>...]) (RE)DEFINE A PACKAGE

Macro in common.lsp. Use to define a package, or redefine a package.

<package> the name of the package to (re)define
<option> any one or more of the following, none evaluated, applied in this order:
(:shadow <symbol>...)
one or more symbols to shadow, as in function SHADOW
(:shadowing-import-from <symbol>...)
one or more symbols to shadow, as in function SHADOWING-IMPORT
(:use <package>...)
one or more packages to "use", as in function USE-PACKAGE
(:import-from <package> <symbol>...)
one or more symbols to import from the package, as in function IMPORT
(:intern <symbol>...)
one or more symbols to be located or created in this package, as in function INTERN
(:export <symbol>...)
one or more symbols to be exported from this package, as in function EXPORT
returns the new or redefined package

(delete-package <package>) DELETE A PACKAGE

Deletes a package by uninterning all its symbols and removing the package.

<package> package to delete
returns T if successful

(do-symbols (<var> [<package> [<result>]]) <expr>...)) ITERATE OVER SYMBOLS
(do-external-symbols (<var> [<package> [<result>]]) <expr>...)
(do-all-symbols (<var> [<result>]) <expr>...)

Implemented as macros in common.lsp. DO-SYMBOLS iterates over all symbols in a single package, DO-EXTERNAL-SYMBOLS iterates only over the external symbols, and DO-ALL-SYMBOLS iterates over all symbols in all packages.

| | |
|---|---|
| <var> | variable to bind to symbol |
| <package> | the package to search |
| <result> | a single result form |
| <expr> | expressions to evaluate (implicit tag-body) |
| returns | result of result form, or NIL if not specified |

(export <symbols> [<package>]) DECLARE EXTERNAL SYMBOLS

| | |
|---|---|
| <symbols> | symbols to declare as external |
| <package> | package symbol is in |
| returns | T |

(find-all-symbols <string>) FIND SYMBOLS WITH SPECIFIED NAME

| | |
|---|---|
| <string> | string or symbol (if latter, print name string is used) |
| returns | list of all symbols having that print-name |

(find-package <package>) FIND PACKAGE WITH SPECIFIED NAME

| | |
|---|---|
| <package> | package to find |
| returns | package with name or nickname <package>, or NIL if not found |

(find-symbol <string> [<package>]) LOOK UP A SYMBOL

| | |
|---|---|
| <string> | print name to search for |
| <package> | package to search in |
| returns | two values, the first being the symbol, and the second being :internal if the symbol is internal in the package, :external if it is external, or :inherited if it is inherited via USE-PACKAGE. If the symbol was not found, then both return values are NIL. |

(import <symbols> [<package>]) IMPORT SYMBOLS INTO A PACKAGE

| | |
|---|---|
| <symbols> | symbols to import (fully qualified names) |
| <package> | package to import symbols into |
| returns | T |

(in-package <package>) SET CURRENT PACKAGE

FSUBR which sets the current package until next call or end of current LOAD.

| | |
|---|---|
| <package> | the package to enter |
| returns | the package |

(list-all-packages) GET ALL PACKAGE NAMES

| | |
|---|---|
| returns | list of all currently existing packages |

(make-package <package> &key :nicknames :use) MAKE A NEW PACKAGE

| | |
|---|---|
| <package> | name of new package to create |
| :nicknames | list of package nicknames |
| :use | list of packages to use (as in USE-PACKAGE) |
| returns | the new package |

(package-name <package>) GET PACKAGE NAME STRING
<package> package name
returns package name string

(package-nicknames <package>) GET PACKAGE NICKNAME STRINGS
<package> package name
returns list of package nickname strings

(package-obarray <package> [<external>]) GET AN OBARRAY
<package> package to use
<external> non-nil for external obarray (default), else internal obarray
returns the obarray (array of lists of symbols in package)

(package-shadowing-symbols <package>) GET LIST OF SHADOWING SYMBOLS
<package> the package
returns list of shadowing symbols in package

(package-use-list <package>) GET PACKAGES USED BY A PACKAGE
<package> the package
returns list of packages used by this package (as in USE-PACKAGE)

(package-used-by-list <package>) GET PACKAGES THAT USE THIS PACKAGE
<package> the package
returns list of packages that use this package (as in USE-PACKAGE)

(package-valid-p <package>) IS THIS A GOOD PACKAGE?
<package> object to check
returns T if a valid package, else NIL

(rename-package <package> <new> [<nick>]) RENAME A PACKAGE
<package> original package
<new> new package name (may be same as original name)
<nick> list of new package nicknames
returns the new package

(shadow <symbols> [<package>]) MAKE SHADOWING SYMBOLS
If a symbol is not already in the package, it is interned. The symbol is placed in the shadowing symbols list for the package.
<symbols> the symbol or symbols to shadow
<package> package to put symbols in
returns T

(shadowing-import <symbols> [<package>]) IMPORT SYMBOLS AND SHADOW
If a symbol exists in the package, it is first uninterned. The symbol is imported, and then made shadowing.
<symbols> the symbol or symbols to import and shadow
<package> package to put symbols in
returns T

(symbol-package <symbol>) FIND THE PACKAGE OF A SYMBOL

| | |
|---|---|
| <symbol> | the symbol |
| returns | the home package of the symbol, or NIL if none |

(unexport <symbols> [<package>]) MAKE SYMBOLS INTERNAL TO PACKAGE

| | |
|---|---|
| <symbols> | symbol or symbols to make internal |
| <package> | package for symbols |
| returns | T |

(unuse-package <pkgs> [<package>]) REMOVE PACKAGES FROM USE LIST

| | |
|---|---|
| <pkgs> | A single package or list of packages |
| <package> | Package in which to un-use packages (default is current package) |
| returns | T |

(use-package <pkgs> [<package>]) ADD PACKAGES TO USE LIST

| | |
|---|---|
| <pkgs> | A single package or list of packages |
| <package> | Package in which to use packages in (default is current package) |
| returns | T |

## PROPERTY LIST FUNCTIONS

Note that property names are not limited to symbols. All functions handle a symbol's property lists except for GETF and REMF which work with any property list.

(get <sym> <prop> [<dflt>]) GET THE VALUE OF A SYMBOL'S PROPERTY

Use as a place form (with SETF) to add or change properties.

| | |
|---|---|
| <sym> | the symbol |
| <prop> | the property name |
| <dflt> | value to return if property not found, default is NIL |
| returns | the property value or <dflt> if property doesn't exist. |

(getf <place> <prop> [<dflt>]) GET THE VALUE OF A PROPERTY

Use GETF as a place form with SETF to add or change properties. (NOTE--when used with SETF, <place> must be a valid place form. It gets executed twice, contrary to Common Lisp standard.)

| | |
|---|---|
| <place> | where the property list is stored |
| <prop> | the property name |
| <dflt> | value to return if property not found, default is NIL |
| returns | the property value or <dflt> if property doesn't exist. |

(putprop <sym> <val> <prop>) PUT A PROPERTY ONTO A PROPERTY LIST

Modern practice is to use (SETF (GET...)...) rather than PUTPROP.

| | |
|---|---|
| <sym> | the symbol |
| <val> | the property value |
| <prop> | the property name |
| returns | the property value |

(remf <place> <prop>) DELETE A PROPERTY

Defined as a macro in COMMON.LSP

| | |
|---|---|
| <place> | where the property list is stored |
| <prop> | the property name |
| returns | T if property existed, else NIL |

(remprop <sym> <prop>) DELETE A SYMBOL'S PROPERTY

| | |
|---|---|
| <sym> | the symbol |
| <prop> | the property name |
| 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 &key :size :test) MAKE A HASH TABLE

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

(gethash <key> <table> [<def>]) EXTRACT FROM HASH TABLE

May be used as place form.

| | |
|---|---|
| <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 <key> <table>) DELETE FROM HASH TABLE

| | |
|---|---|
| <key> | hash key |
| <table> | hash table |
| returns | T if deleted, NIL if not in table |

(clrhash <table>) CLEAR THE HASH TABLE

| | |
|---|---|
| <table> | hash table |
| returns | NIL, all entries cleared from table |

(hash-table-count <table>) NUMBER OF ENTRIES IN HASH TABLE

| | |
|---|---|
| <table> | hash table |
| returns | integer number of entries in table |

(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 <array> <n>) GET THE NTH ELEMENT OF AN ARRAY

May be used as a place form

<array> the array (or string)

<n> the array index (fixnum, zero based)

returns the value of the array element

(make-array <size> &key :initial-element :initial-contents) MAKE A NEW ARRAY

<size> the size of the new array (fixnum)

:initial-element

value to initialize all array elements, default NIL

:initial-contents

sequence used to initialize all array elements, consecutive sequence elements are used for each array element. The length of the sequence must be the same as the size of the array

returns the new array

(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 <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 arguement sequences |

(elt <expr> <n>) GET THE NTH ELEMENT OF A SEQUENCE
May be used as a place form

| | |
|---|---|
| <expr> | the sequence |
| <n> | the index of element to return |
| returns | the element if the index is in bounds, otherwise error |

(map <type> <fcn> <expr> ...) APPLY FUNCTION TO SUCCESSIVE ELEMENTS
(map-into <target> <fcn> [<expr> ...])

| | |
|---|---|
| <type> | result type, one of CONS, LIST, ARRAY, STRING, or NIL |
| <target> | destination sequence to modify |
| <fcn> | the function or function name |
| <expr> | a sequence for each argument of the function |
| returns | a new sequence of type <type> for MAP, and <target> for MAP-INTO. |

(every <fcn> <expr> ...) APPLY FUNCTION TO ELEMENTS UNTIL FALSE
(notevery <fcn> <expr> ...)

| | |
|---|---|
| <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 <fcn> <expr> ...) APPLY FUNCTION TO ELEMENTS UNTIL TRUE
(notany <fcn> <expr> ...)

| | |
|---|---|
| <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 <expr>) FIND THE LENGTH OF A SEQUENCE
Note that a circular list causes an error. To detect a circular list, use LIST-LENGTH.

| | |
|---|---|
| <expr> | the list, vector or string |
| returns | the length of the list, vector or string |

(reverse <expr>) REVERSE A SEQUENCE
(nreverse <expr>) DESTRUCTIVELY REVERSE A SEQUENCE

| | |
|---|---|
| <expr> | the sequence to reverse |
| returns | a new sequence in the reverse order |

(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> |

(sort <seq> <test> &key :key) DESTRUCTIVELY SORT A SEQUENCE
(stable-sort <seq> <test> &key :key) STABLE DESTRUCTIVE SORT

| | |
|---|---|
| <seq> | the sequence to sort |
| <test> | the comparison function, must return T only if its first argument is strictly to the left of its second argument. |
| :key | function to apply to comparison function arguments (defaults to identity) |
| returns | the sorted sequence |

(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 <expr> <seq> &key :test :test-not :key :start :end :count :from-end)
REMOVE ELEMENTS FROM A SEQUENCE
(remove-if <test> <seq> &key :key :start :end :count :from-end)
REMOVE ELEMENTS THAT PASS TEST
(remove-if-not <test> <seq> &key :key :start :end :count :from-end)
REMOVE ELEMENTS THAT FAIL TEST

| | |
|---|---|
| <expr> | the element to remove |
| <test> | the test predicate, applied to each <seq> element in turn |
| <seq> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <seq> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <seq>) |
| :count | maximum number of elements to remove, negative values treated as zero, NIL same as default -- unlimited. |
| :from-end | if non-nil, behaves as though elements are removed from right end. This only has an affect when :count is used. |
| returns | copy of sequence with matching/non-matching expressions removed |

(count <expr> <seq> &key :test :test-not :key :start :end :from-end)
COUNT MATCHING ELEMENTS IN A SEQUENCE
(count-if <test> <seq> &key :key :start :end :from-end)
COUNT ELEMENTS THAT PASS TEST
(count-if-not <test> <seq> &key :key :start :end :from-end)
COUNT ELEMENTS THAT FAIL TEST

| | |
|---|---|
| <expr> | element to count |
| <test> | the test predicate, applied to each <seq> element in turn |
| <seq> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <seq> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <seq>) |
| :from-end | this argument is ignored |
| returns | count of matching/non-matching elements |

(find <expr> <seq> &key :test :test-not :key :start :end :from-end)
FIND FIRST MATCHING ELEMENT IN SEQUENCE
(find-if <test> <seq> &key :key :start :end :from-end)
FIND FIRST ELEMENT THAT PASSES TEST
(find-if-not <test> <seq> &key :key :start :end :from-end)
FIND FIRST ELEMENT THAT FAILS TEST

| | |
|---|---|
| <expr> | element to search for |
| <test> | the test predicate, applied to each <seq> element in turn |
| <seq> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <seq> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <seq>) |
| :from-end | if non-nil search is done for last element |
| returns | first matching/non-matching element of sequence, or NIL |

(position <expr> <seq> &key :test :test-not :key :start :end :from-end)
FIND POSITION OF FIRST MATCHING ELEMENT IN SEQUENCE
(position-if <test> <seq> &key :key :start :end :from-end)
FIND POSITION OF FIRST ELEMENT THAT PASSES TEST
(position-if-not <test> <seq> &key :key :start :end :from-end)
FIND POSITION OF FIRST ELEMENT THAT FAILS TEST

| | |
|---|---|
| <expr> | element to search for |
| <test> | the test predicate, applied to each <seq> element in turn |
| <seq> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <seq> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <seq>) |
| :from-end | if non-nil search is made for last element |
| returns | position of first matching/non-matching element of sequence, or NIL |

(delete <expr> <seq> &key :key :test :test-not :start :end :count :from-end)
DELETE ELEMENTS FROM A SEQUENCE
(delete-if <test> <seq> &key :key :start :end :count :from-end)
DELETE ELEMENTS THAT PASS TEST
(delete-if-not <test> <seq> &key :key :start :end :count :from-end)
DELETE ELEMENTS THAT FAIL TEST

| | |
|---|---|
| <expr> | the element to delete |
| <test> | the test predicate, applied to each <seq> element in turn |
| <seq> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <seq> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <seq>) |
| :count | maximum number of elements to remove, negative values treated as zero, NIL same as default -- unlimited. |
| :from-end | if non-nil, behaves as though elements are removed from right end. This only has an affect when :count is used. |
| returns | <seq> with the matching/non-matching expressions deleted |

(substitute <r> <e> <s> &key :key :test :test-not :start :end :count :from-end)
SUBSTITUTE ELEMENTS IN A SEQUENCE
(substitute-if <r> <test> <s> &key :key :start :end :count :from-end)
SUBSTITUTE ELEMENTS THAT PASS TEST
(substitute-if-not <r> <test> <s> &key :key :start :end :count :from-end)
SUBSTITUTE ELEMENTS THAT FAIL TEST

| | |
|---|---|
| <r> | the replacement expression |
| <e> | the element to replace |
| <test> | the test predicate, applied to each <s> element in turn |
| <s> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <s> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <s>) |
| :count | maximum number of elements to remove, negative values treated as zero, NIL same as default -- unlimited. |
| :from-end | if non-nil, behaves as though elements are removed from right end. This only has an affect when :count is used. |
| returns | copy of <s> with the matching/non-matching expressions substituted |

(nsubstitute <r> <e> <s> &key :key :test :test-not :start :end :count :from-end)
DESTRUCTIVELY SUBSTITUTE ELEMENTS IN A SEQUENCE
(nsubstitute-if <r> <test> <s> &key :key :start :end :count :from-end)
DESTRUCTIVELY SUBSTITUTE ELEMENTS THAT PASS TEST
(nsubstitute-if-not <r> <test> <s> &key :key :start :end :count :from-end)
DESTRUCTIVELY SUBSTITUTE ELEMENTS THAT FAIL TEST

| | |
|---|---|
| <r> | the replacement expression |
| <e> | the element to replace |
| <test> | the test predicate, applied to each <s> element in turn |
| <s> | the sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to each <s> element (defaults to identity) |
| :start | starting index |
| :end | index of end+1, or NIL for (length <s>) |
| :count | maximum number of elements to remove, negative values treated as zero, NIL same as default -- unlimited. |
| :from-end | if non-nil, behaves as though elements are removed from right end. This only has an affect when :count is used. |
| returns | <s> with the matching/non-matching expressions substituted |

(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 <seq> &key :test :test-not :key :start :end)
MAKE SEQUENCE WITH DUPLICATES REMOVED
(delete-duplicates <seq> &key :test :test-not :key :start :end)
DELETE DUPLICATES FROM SEQUENCE

Delete-duplicates defined in common2.lsp.

| | |
|---|---|
| <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, or <seq> with duplicates deleted (destructive). |

(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 <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 |

(make-sequence <type> <size> &key :initial-element) MAKE A SEQUENCE

Defined in common2.lsp.

| | |
|---|---|
| <type> | type of sequence to create: CONS LIST ARRAY or STRING |
| <size> | size of sequence (non-negative integer) |
| :initial-element | initial value of all elements in sequence |
| returns | the new sequence |

(copy-seq <seq>) COPY A SEQUENCE

Defined in common2.lsp

| | |
|---|---|
| <seq> | sequence to copy |
| returns | copy of the sequence, sequence elements are eq those in the original sequence. |

(merge <type> <seq1> <seq2> <pred> &key :key) MERGE TWO SEQUENCES

Defined in common2.lsp. Non-destructive, although may be destructive in Common Lisp.

| | |
|---|---|
| <type> | type of result sequence: CONS LIST ARRAY or STRING |
| <seq1> | first sequence to merge |
| <seq2> | second sequence to merge |
| <pred> | function of two arguments which returns true if its first argument should precede its second |
| :key | optional function to apply to each sequence element before applying predicate function (defaults to identity) |
| returns | new sequence containing all the elements of seq1 (in order) merged with all the elements of seq2, according to the predicate function |

(mismatch <s1> <s2> &key :test :test-not :key :start1 :end1 :start2 :end2)

FIND DIFFERENCE BETWEEN TWO SEQUENCES

Defined in common2.lsp.

| | |
|---|---|
| <s1> | first sequence |
| <s2> | second sequence |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to to each sequence element before applying test function (defaults to identity) |
| :start1 | starting index in <s1> |
| :end1 | index of end+1 in <s1> or NIL for end of sequence |
| :start2 | starting index in <s2> |
| :end2 | index of end+1 in <s2> or NIL for end of sequence |
| returns | integer index of first mismatch in s1, or NIL if no mismatch |

# LIST FUNCTIONS

(car <expr>) RETURN THE CAR OF A LIST NODE

May be used as a place form.

| | |
|---|---|
| <expr> | the list node |
| returns | the car of the list node |

(cdr <expr>) RETURN THE CDR OF A LIST NODE

May be used as a place form.

| | |
|---|---|
| <expr> | the list node |
| returns | the cdr of the list node |

(cxxr <expr>) ALL CxxR COMBINATIONS
(cxxxr <expr>) ALL CxxxR COMBINATIONS
(cxxxxr <expr>) ALL CxxxxR COMBINATIONS

May be used as place forms when COMMON2.LSP loaded.

(first <expr>) A SYNONYM FOR CAR
(second <expr>) A SYNONYM FOR CADR
(third <expr>) A SYNONYM FOR CADDR
(fourth <expr>) A SYNONYM FOR CADDDR
(fifth <expr>) FIFTH LIST ELEMENT
(sixth <expr>) SIXTH LIST ELEMENT
(seventh <expr>) SEVENTH LIST ELEMENT
(eighth <expr>) EIGHTH LIST ELEMENT
(ninth <expr>) NINTH LIST ELEMENT
(tenth <expr>) TENTH LIST ELEMENT
(rest <expr>) A SYNONYM FOR CDR

May be used as place forms when COMMON2.LSP loaded. fifth through tenth defined in COMMON2.LSP.

(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 <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 <expr>...) CREATE A LIST OF VALUES
(list* <expr> ... <list>)

| | |
|---|---|
| <expr> | expressions to be combined into a list |
| returns | the new list |

(append <expr>...) APPEND LISTS

| | |
|---|---|
| <expr> | lists whose elements are to be appended |
| returns | the new list |

(revappend <expr1> <expr2>) APPEND REVERSE LIST

Defined in common2.lsp

| | |
|---|---|
| <expr1> | first list |
| <expr2> | second list |
| returns | new list comprised of reversed first list appended to second list |

(list-length <list>) FIND THE LENGTH OF A LIST

| | |
|---|---|
| <list> | the list |
| returns | the length of the list or NIL if the list is circular |

(last <list>) RETURN THE LAST LIST NODE OF A LIST

| | |
|---|---|
| <list> | the list |
| returns | the last list node in the list |

(tailp <sublist> <list>) IS ONE LIST A SUBLIST OF ANOTHER?

Defined in common2.lsp

| | |
|---|---|
| <sublist> | list to search for |
| <list> | list to search in |
| returns | T if sublist is EQ one of the top level conses of list |

(butlast <list> [<n>]) RETURN COPY OF ALL BUT LAST OF LIST
(nbutlast <list> [<n>]) DELETE LAST ELEMENTS OF LIST

nbutlast defined in common2.lsp

| | |
|---|---|
| <list> | the list |
| <n> | count of elements to omit (default 1) |
| returns | copy of list with last element(s) absent, or, for nbutlast, the list with the last elements deleted (destructive). |

(nth <n> <list>) RETURN THE NTH ELEMENT OF A LIST

May be used as a place form

| | |
|---|---|
| <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 <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 <expr> <list> &key :test :test-not :key) FIND AN EXPRESSION IN A LIST
(member-if <test> <list> &key :key) FIND ELEMENT PASSING TEST
(member-if-not <test> <list> &key :key) FIND ELEMENT FAILING TEST

Functions member-if and member-if-not defined in common2.lsp

| | |
|---|---|
| <expr> | the expression to find |
| <test> | the test predicate |
| <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 or element passing/failing the test predicate |

(assoc <expr> <alist> &key :test :test-not :key) FIND AN EXPRESSION IN AN A-LIST
(assoc-if <test> <alist> &key :key) FIND ELEMENT IN A-LIST PASSING TEST
(assoc-if-not <test> <alist> &key :key) FIND ELEMENT IN A-LIST FAILING TEST
(rassoc <expr> <alist> &key :test :test-not :key) FIND AN EXPRESSION IN AN A-LIST
(rassoc-if <test> <alist> &key :key) FIND ELEMENT IN A-LIST PASSING TEST
(rassoc-if-not <test> <alist> &key :key) FIND ELEMENT IN A-LIST FAILING TEST

All functions but assoc defined in common2.lsp. The rassoc functions match the cdrs of the a-list elements while the assoc functions match the cars.

| | |
|---|---|
| <expr> | the expression to find |
| <test> | the test predicate |
| <alist> | the association list |
| :test | the test function (defaults to eql) |
| :test-not | the test function (sense inverted) |
| :key | function to apply to a-list argument (defaults to identity) |
| returns | the alist entry or NIL |

(mapc <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CARS
(mapcar <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CARS
(mapcan <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CARS
(mapl <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CDRS
(maplist <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CDRS
(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 | the first list of arguments (mapc or mapl), a list of the values returned (mapcar or maplist), or list or returned values nconc'd together (mapcan or mapcon). |

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

SUBSTITUTE EXPRESSIONS

SUBST does minimum copying as required by Common Lisp. NSUBST is the destructive version.

| | |
|---|---|
| <to> | the new expression |
| <from> | the old expression (match to part of <expr> using test function |
| <test> | test predicate |
| <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 subtree test function expression argument (defaults to identity) |
| returns | the expression with substitutions |

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

SUBSTITUTE WITH AN A-LIST

SUBLIS does minimum copying as required by Common Lisp. NSUBLIS is the destructive version.

| | |
|---|---|
| <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 subtree test function expression argument (defaults to identity) |
| returns | the expression with substitutions |

(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 |

(make-list <size> &key :initial-element) MAKE A LIST

In file common2.lsp

| | |
|---|---|
| <size> | size of list (non-negative integer) |
| :initial-element | initial value for each element, default NIL |
| returns | the new list |

(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 <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 <tree>) COPY A TREE

In file common.lsp

| | |
|---|---|
| <tree> | a tree structure of cons cells |
| returns | a copy of the tree structure |

(intersection <list1> <list2> &key :test :test-not :key) SET FUNCTIONS
(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 <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>. |

(ldiff <list> <sublist>) GET INITIAL ELEMENTS OF LIST

In file common2.lsp

| | |
|---|---|
| <list> | list to get elements of |
| <sublist> | list to search for in <list> (uses tailp) |
| returns | copy of list up to match with sublist. |

## DESTRUCTIVE LIST FUNCTIONS

Destructive functions that have non-desctructive equivalents are listed in other sections. See also sort, map-into, nreverse, delete, delete-if, delete-if-not, fill, and replace under SEQUENCE FUNCTIONS, setf under SYMBOL FUNCTIONS, and mapcan, mapcon, nbutlast, nsubst, nsubst-if, nsubst-if-not, nsublis, nintersection, nunion, nset-difference, and nset-exclusive-or under LIST FUNCTIONS. Also, setf is a destructive function.

(rplaca <list> <expr>) REPLACE THE CAR OF A LIST NODE

Modern practice is to use (setf (car <list>) <expr>)

| | |
|---|---|
| <list> | the list node |
| <expr> | the new value for the car of the list node |
| returns | the list node after updating the car |

(rplacd <list> <expr>) REPLACE THE CDR OF A LIST NODE

Modern practice is to use (setf (cdr <list>) <expr>)

| | |
|---|---|
| <list> | the list node |
| <expr> | the new value for the cdr of the list node |
| returns | the list node after updating the cdr |

(nconc <list>...) DESTRUCTIVELY CONCATENATE LISTS

| | |
|---|---|
| <list> | lists to concatenate |
| returns | the result of concatenating the lists |

(nreconc <list1> <list2>) DESTRUCTIVELY CONCATENATE LISTS

Defined in common2.lsp

| | |
|---|---|
| <list1> | first list |
| <list2> | second list |
| returns | second list concantenated to the end of the first list, which has been destructively reversed. |

# ARITHMETIC FUNCTIONS

Warning: integer calculations that overflow become floating point values as part of the math extension, but give no error in the base-line Xlisp. Integer calculations cannot overflow when the bignum extension is compiled. On systems with IEEE floating point, the values +INF and -INF result from overflowing floating point calculations.

The math extension option adds complex numbers, new functions, and additional functionality to some existing functions. The bignum extension, in addition, adds ratios, bignums, new functions and additional functionality to some existing functions. Because of the size of the extensions, and the performance loss they entail, some users may not wish to include bignums, or bignums and math. This section documents the math functions both with and without the extensions.

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 (bignum extension) 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. Rational complex numbers with zero imaginary parts are reduced to integers.

(truncate <expr> <denom>) TRUNCATES TOWARD ZERO
(round <expr> <denom>) ROUNDS TOWARD NEAREST EVEN INTEGER
(floor <expr> <denom>) TRUNCATES TOWARD NEGATIVE INFINITY
(ceiling <expr> <denom>) TRUNCATES TOWARD INFINITY

Round, floor, and ceiling, and the second argument of truncate, are 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, and, as a second return value, the remainder of the operation, defined as expr - result*denom. The type is flonum if either argument is flonum, otherwise it is rational. |

(float <expr>) CONVERTS AN INTEGER TO A FLOATING POINT NUMBER

| | |
|---|---|
| <expr> | the real number |
| returns | the number as a flonum |

(rational <expr>) CONVERTS A REAL NUMBER TO A RATIONAL

Rational numbers are returned as is. Part of the bignum extension.

| | |
|---|---|
| <expr> | the real number |
| returns | the number as a ratio or integer. |

(+ [<expr>...]) ADD A LIST OF NUMBERS

With no arguments returns addition identity, 0 (integer)

| | |
|---|---|
| <expr> | the numbers |
| returns | the result of the addition |

(- <expr>...) SUBTRACT A LIST OF NUMBERS OR NEGATE A SINGLE NUMBER

| | |
|---|---|
| <expr> | the numbers |
| returns | the result of the subtraction |

(* [<expr>...]) MULTIPLY A LIST OF NUMBERS

With no arguments returns multiplication identity, 1

| | |
|---|---|
| <expr> | the numbers |
| returns | the result of the multiplication |

(/ <expr>...) DIVIDE A LIST OF NUMBERS OR INVERT A SINGLE NUMBER

With the bignum extension, division of integer numbers results in a rational quotient, rather than integer. To perform integer division, use TRUNCATE.

| | |
|---|---|
| <expr> | the numbers |
| returns | the result of the division |

(1+ <expr>) ADD ONE TO A NUMBER

| | |
|---|---|
| <expr> | the number |
| returns | the number plus one |

(1- <expr>) SUBTRACT ONE FROM A NUMBER

| | |
|---|---|
| <expr> | the number |
| returns | the number minus one |

(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 <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 <expr>...) THE SMALLEST OF A LIST OF NUMBERS

| | |
|---|---|
| <expr> | the real numbers |
| returns | the smallest number in the list |

(max <expr>...) THE LARGEST OF A LIST OF NUMBERS

| | |
|---|---|
| <expr> | the real numbers |
| returns | the largest number in the list |

(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 <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. |

(float-sign <expr1> [<expr2>]) APPLY SIGN TO A NUMBER

Defined in common2.lsp

| | |
|---|---|
| <expr1> | the real number |
| <expr2> | another real number, defaults to 1.0 |
| returns | the number <expr2> with the sign of <expr1> |

(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 <n>...) COMPUTE THE LEAST COMMON MULTIPLE

Part of math extension. A result which would be larger than the largest integer causes an error.

| | |
|---|---|
| <n> | The number(s) (integer) |
| returns | the least common multiple |

(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 [<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 <expr>) COMPUTE THE SINE OF A NUMBER
(cos <expr>) COMPUTE THE COSINE OF A NUMBER
(tan <expr>) COMPUTE THE TANGENT OF A NUMBER
(asin <expr>) COMPUTE THE ARC SINE OF A NUMBER
(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 <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 <expr>) COMPUTE THE HYPERBOLIC SINE OF A NUMBER
(cosh <expr>) COMPUTE THE HYPERBOLIC COSINE OF A NUMBER
(tanh <expr>) COMPUTE THE HYPERBOLIC TANGENT OF A NUMBER
(asinh <expr>) COMPUTE THE HYPERBOLIC ARC SINE OF A NUMBER
(acosh <expr>) COMPUTE THE HYPERBOLIC ARC COSINE OF A NUMBER
(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 <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 an integer, then the result type is the same as the type of x. |

(exp <x-expr>) COMPUTE E TO THE X POWER

| | |
|---|---|
| <x-expr> | the floating point number |
| returns | e to the x power |

(cis <x-expr>) COMPUTE COSINE + I SINE

Defined in common.lsp

| | |
|---|---|
| <x-expr> | the number |
| returns | e to the ix power |

(log <expr> [<base>]) COMPUTE THE LOGRITHM

Part of the math extension

| | |
|---|---|
| <expr> | the number |
| <base> | the base, default is e |
| returns | log base <base> of <expr> |

(sqrt <expr>) COMPUTE THE SQUARE ROOT OF A NUMBER

| | |
|---|---|
| <expr> | the number |
| returns | the square root of the number |

(isqrt <expr>) COMPUTER THE INTEGER SQUARE ROOT OF A NUMBER

Defined in common2.lsp

| | |
|---|---|
| <expr> | non-negative integer |
| returns | the integer square root, either exact or the largest integer less than the exact value. |

(numerator <expr>) GET THE NUMERATOR OF A NUMBER

Part of bignum extension

| | |
|---|---|
| <expr> | rational number |
| returns | numerator of number (number if integer) |

(denominator <expr>) GET THE DENOMINATOR OF A NUMBER

Part of bignum extension

| | |
|---|---|
| <expr> | rational number |
| returns | denominator of number (1 if integer) |

(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 <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 <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 <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 <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>))

(< <n1> <n2>...) TEST FOR LESS THAN
(<= <n1> <n2>...) TEST FOR LESS THAN OR EQUAL TO
(= <n1> <n2>...) TEST FOR EQUAL TO
(/= <n1> <n2>...) TEST FOR NOT EQUAL TO
(>= <n1> <n2>...) TEST FOR GREATER THAN OR EQUAL TO
(> <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

Integers are treated as two's complement, which can cause what appears to be strange results when negative numbers are supplied as arguments.

(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 [<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 [<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

(logeqv [<expr>...]) THE BITWISE EQUIVALENCE OF A LIST OF INTEGERS
With no arguments returns identity -1
<expr> the integers
returns the result of the equivalence operation

(lognand <expr1> <expr2>) BITWISE LOGICAL FUNCTIONS
(logandc1 <expr1> <expr2>)
(logandc2 <expr1> <expr2>)
(lognor <expr1> <expr2>)
(logorc1 <expr1> <expr2>)
(logorc2 <expr1> <expr2>)
Part of the bignums extension, the remaining logical functions of two integers.
<expr1> the first integer
<expr2> the second integer
returns lognand: (lognot (logand <expr1> <expr2>))
logandc1: (logand (lognot <expr1>) <expr2>)
logandc2: (logand <expr1> (lognot <expr2>))
lognor: (lognot (logor <expr1> <expr2>))
logorc1: (logor (lognot <expr1>) <expr2>)
logorc2: (logor <expr1> (lognot <expr2>))

(lognot <expr>) THE BITWISE NOT OF A INTEGER
<expr> the integer
returns the bitwise inversion of integer

(logtest <expr1> <expr2>) TEST BITWISE AND OF TWO INTEGERS
Defined in common.lsp when bignum extension not loaded
<expr1> the first integer
<expr2> the second integer
returns T if the result of the and operation is non-zero, else NIL

(logbitp <pos> <expr>) TEST BIT OF INTEGER

Part of bignums extension

| | |
|---|---|
| <pos> | non-negative fixnum bit position, as in (expt 2 <pos>) |
| <expr> | integer to test |
| returns | T if the bit is "1", else NIL. |

(logcount <expr>) COUNT BITS IN AN INTEGER

Part of bignums extension

| | |
|---|---|
| <expr> | integer |
| returns | if <expr> is negative, returns the number of 0 bits, else returns the number of 1 bits |

(integer-length <expr>) CALCULATE LENGTH OF AN INTEGER

Part of bignums extension

| | |
|---|---|
| <expr> | integer |
| returns | the minimum number of bits necessary to represent the integer, excluding any sign bit. |

(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 |

(byte <size> <pos>) CREATE A BYTE SPECIFIER
(byte-size <spec>) GET SPECIFIER SIZE FIELD
(byte-position <spec>) GET SPECIFIER POSITION FIELD

Defined in common2.lsp. A "byte specifier" is implemented as a CONS cell with the CAR being the size and the CDR being the position. These functions are aliases for CONS, CAR, and CDR, respectively.

| | |
|---|---|
| <size> | size of byte field (non-negative integer) |
| <pos> | starting position of byte field (non-negative integer), which is position with least bit weight |
| <spec> | byte specifier (a CONS cell) |
| returns | BYTE returns the specifier, BYTE-SIZE returns the size field, and BYTE-POSITION returns the starting position |

(ldb <spec> <int>) LOAD BYTE

Defined in common2.lsp. LDB can be used with SETF, in which case it performs a STB followed by a SETF into the field.

| | |
|---|---|
| <spec> | specifier of byte to extract |
| <int> | integer to extract byte from |
| returns | the extracted byte, a non-negative integer |

(ldb-test <spec> <int>) TEST A BYTE

Defined in common2.lsp

| | |
|---|---|
| <spec> | specifier of byte to test |
| <int> | integer containing byte to test |
| returns | T if byte is zero, else NIL |

(mask-field <spec> <int>) EXTRACT UNDER MASK

Defined in common2.lsp. MASK-FIELD can be used with SETF, in which case it performs a DEPOSIT-FIELD followed by a SETF into the field.

| | |
|---|---|
| <spec> | specified byte to extract |
| <int> | integer to extract byte from |
| returns | the extracted byte in the same bit position as it was in <int> |

(dpb <new> <spec> <int>) DEPOSIT BYTE

Defined in common2.lsp

| | |
|---|---|
| <new> | integer byte to insert |
| <spec> | specifier of position and size of byte |
| <int> | integer to insert byte into |
| returns | <int> with <new> in the bit positions specified by <spec> |

(deposit-field <new> <spec> <int>) INSERT UNDER MASK

Defined in common2.lsp

| | |
|---|---|
| <new> | integer containing byte field to insert |
| <spec> | specifier of position and size of byte |
| <int> | integer to insert byte into |
| returns | <new> at <spec> replacing bits at <spec> in <int> |

## STRING FUNCTIONS

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

| (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 <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 <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 <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 <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 <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 |

| (string-capitalize <str> &key :start :end) | CAPITALIZE STRING |
|---|---|
| <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 with each word having an initial uppercase letter and following lowercase letters |

| (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 <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) |

(nstring-capitalize <str> &key :start :end) CAPITALIZE STRING

| | |
|---|---|
| <str> | the string |
| :start | the starting offset |
| :end | the ending offset + 1 or NIL for end of string |
| returns | the string with each word having an initial uppercase letter and following lowercase letters (not a copy) |

(make-string <size> &key :initial-element) MAKE A STRING

Defined in common2.lsp.

| | |
|---|---|
| <size> | size of string (non-negative integer) |
| :initial-element | initial value of all characters in the string |
| returns | the new string |

(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< <str1> <str2> &key :start1 :end1 :start2 :end2)
(string<= <str1> <str2> &key :start1 :end1 :start2 :end2)
(string= <str1> <str2> &key :start1 :end1 :start2 :end2)
(string/= <str1> <str2> &key :start1 :end1 :start2 :end2)
(string>= <str1> <str2> &key :start1 :end1 :start2 :end2)
(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<br>others: If predicate is true then number of initial matching characters, else NIL |

Note: case is significant with these comparison functions.

(string-lessp <str1> <str2> &key :start1 :end1 :start2 :end2)
(string-not-greaterp <str1> <str2> &key :start1 :end1 :start2 :end2)
(string-equal <str1> <str2> &key :start1 :end1 :start2 :end2)
(string-not-equal <str1> <str2> &key :start1 :end1 :start2 :end2)
(string-not-lessp <str1> <str2> &key :start1 :end1 :start2 :end2)
(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 -- all uppercase characters are converted to lowercase before being compared.

# CHARACTER FUNCTIONS

(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 |

(alphanumericp <chr>) IS THIS CHARACTER ALPHANUMERIC?

| | |
|---|---|
| <chr> | the character |
| returns | true if the character is alphabetic or numeric, NIL otherwise |

(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 <chr>) IS THIS A LOWER CASE CHARACTER?

| | |
|---|---|
| <chr> | the character |
| returns | true if the character is lower case, NIL otherwise |

(alpha-char-p <chr>) IS THIS AN ALPHABETIC CHARACTER?

| | |
|---|---|
| <chr> | the character |
| returns | true if the character is alphabetic, NIL otherwise |

(both-case-p <chr>) IS THIS AN ALPHABETIC (EITHER CASE) CHARACTER?

| | |
|---|---|
| <chr> | the character |
| returns | true if the character is available in both cases, NIL otherwise |

(digit-char-p <chr>[<radix>]) IS THIS A DIGIT CHARACTER?

| | |
|---|---|
| <chr> | the character |
| <radix> | the radix (default 10) |
| returns | the digit weight if character is a digit, NIL otherwise |

(char-code <chr>) GET THE ASCII CODE OF A CHARACTER

| | |
|---|---|
| <chr> | the character |
| returns | the ASCII character code (integer, parity bit stripped) |

(code-char <code>) GET THE CHARACTER WITH A SPECFIED ASCII CODE

| | |
|---|---|
| <code> | the ASCII code (integer, range 0-127) |
| returns | the character with that code or NIL |

(char-upcase <chr>) CONVERT A CHARACTER TO UPPER CASE

| | |
|---|---|
| <chr> | the character |
| returns | the upper case version of the character, if one exists, otherwise returns the character |

(char-downcase <chr>) CONVERT A CHARACTER TO LOWER CASE

| | |
|---|---|
| <chr> | the character |
| returns | the lower case version of the character, if one exists, otherwise returns the character |

(digit-char <n>[<radix>]) CONVERT A DIGIT WEIGHT TO A DIGIT

| | |
|---|---|
| <n> | the digit weight (integer) |
| <radix> | the radix (default 10) |
| returns | the digit character or NIL |

(char-int <chr>) CONVERT A CHARACTER TO AN INTEGER

| | |
|---|---|
| <chr> | the character |
| returns | the ASCII character code (range 0-255) |

(int-char <int>) CONVERT AN INTEGER TO A CHARACTER

| | |
|---|---|
| <int> | the ASCII character code (treated modulo 256) |
| returns | the character with that code |

(character <expr>) CREATE A CHARACTER

Defined in common2.lsp

| | |
|---|---|
| <expr> | single character symbol, string, or integer |
| returns | <expr> converted into a character |

(char-name <chr>) CHARACTER PRINT NAME

Defined in common2.lsp

| | |
|---|---|
| <chr> | the character |
| returns | string which is the name of the character, or NIL if no name |

(char< <chr1> <chr2>...)
(char<= <chr1> <chr2>...)
(char= <chr1> <chr2>...)
(char/= <chr1> <chr2>...)
(char>= <chr1> <chr2>...)
(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 <chr1> <chr2>...)
(char-not-greaterp <chr1> <chr2>...)
(char-equal <chr1> <chr2>...)
(char-not-equal <chr1> <chr2>...)
(char-not-lessp <chr1> <chr2>...)
(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 -- all uppercase characters are converted to lowercase before the comparison.

## 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.

```
(defstruct name [<comment>] <slot-desc>...)
```
or
```
(defstruct (name <option>...) [<comment>] <slot-desc>...)
```

fsubr

| | |
|---|---|
| <name> | the structure name symbol (quoted) |
| <option> | option description (quoted) |
| <comment> | comment string (ignored) |
| <slot-desc> | slot descriptions (quoted) |
| returns | the structure name |

The recognized options are:

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

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.

The optional :PRINT-FUNCTION overrides the default #S notation. The function must take three arguments, the structure instance, the stream, and the current printing depth.

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>)
place form (foo-gag <expr>)
(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 starting on page 21.

(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 <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 <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 via (setf (send <object> :<ivar>) <value>).
- (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.
- returns the new class object

(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 selector name (quoted)
- <fargs> Formal argument list. Leading "self" is implied (quoted)
- <expr> Expressions constituting body of method (quoted)
- returns the class object.

(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) |
| returns | the instance object |

(tracemethod <class> [<sel>]) ADD A METHOD TO THE TRACE LIST

defined in class.lsp

| | |
|---|---|
| <class> | Class containing method to trace |
| <sel> | Message selector of method to trace, if absent then trace all methods defined in <class>. |
| returns | the trace list |

(untracemethod [<class> [<sel>]])) REMOVE A METHOD FROM THE TRACE LIST

defined in class.lsp

| | |
|---|---|
| <class> | Class containing method to remove, if absent remove all methods of all classes. |
| <sel> | Message selector of method to remove, if absent then remove all methods of <class>. |
| returns | the trace list |

## PREDICATE FUNCTIONS

(atom <expr>) IS THIS AN ATOM?
<expr> the expression to check
returns t if the value is an atom, NIL otherwise

(symbolp <expr>) IS THIS A SYMBOL?
<expr> the expression to check
returns t if the expression is a symbol, NIL otherwise

(numberp <expr>) IS THIS A NUMBER?
<expr> the expression to check
returns t if the expression is a number, NIL otherwise

(null <expr>) IS THIS AN EMPTY LIST?
<expr> the list to check
returns t if the list is empty, NIL otherwise

(not <expr>) IS THIS FALSE?
<expr> the expression to check
return t if the value is NIL, NIL otherwise

(listp <expr>) IS THIS A LIST?
<expr> the expression to check
returns t if the value is a cons or NIL, NIL otherwise

(endp <list>) IS THIS THE END OF A LIST?
<list> the list
returns t if the value is NIL, NIL otherwise

(consp <expr>) IS THIS A NON-EMPTY LIST?
<expr> the expression to check
returns t if the value is a cons, NIL otherwise

(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.

(specialp <expr>) IS THIS A SPECIAL SYMBOL?
<expr> the expression to check
returns t if the value is a symbol which is SPECIAL, NIL otherwise.

(integerp <expr>) IS THIS AN INTEGER?
<expr> the expression to check
returns t if the value is an integer, NIL otherwise

(floatp <expr>) IS THIS A FLOAT?
<expr> the expression to check
returns t if the value is a float, NIL otherwise

(rationalp <expr>) IS THIS A RATIONAL NUMBER?
Part of bignum extension.
<expr> the expression to check
returns t if the value is rational (integer or ratio), NIL otherwise

(realp <expr>) IS THIS A REAL NUMBER?
Defined in common2.lsp
<expr> the expression to check
returns t if the value is rational or float, NIL otherwise

(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 <expr>) IS THIS A STRING?
<expr> the expression to check
returns t if the value is a string, NIL otherwise

(characterp <expr>) IS THIS A CHARACTER?
<expr> the expression to check
returns t if the value is a character, NIL otherwise

(arrayp <expr>) IS THIS AN ARRAY?
<expr> the expression to check
returns t if the value is an array, NIL otherwise

(array-in-bounds-p <expr> <index>) IS ARRAY INDEX IN BOUNDS?
Defined in common2.lsp
<expr> the array
<index> index to check
returns t if index is in bounds for the array, NIL otherwise

(streamp <expr>) IS THIS A STREAM?
<expr> the expression to check
returns t if the value is a stream, NIL otherwise

(open-stream-p <stream>) IS STREAM OPEN?
<stream> the stream
returns t if the stream is open, NIL otherwise

(input-stream-p <stream>) IS STREAM READABLE?
<stream> the stream
returns t if stream is readable, NIL otherwise

(output-stream-p <stream>) IS STREAM WRITABLE?
<stream> the stream
returns t if stream is writable, NIL otherwise

(objectp <expr>) IS THIS AN OBJECT?
<expr> the expression to check
returns t if the value is an object, NIL otherwise

(classp <expr>) IS THIS A CLASS OBJECT?
<expr> the expression to check
returns t if the value is a class object, NIL otherwise

(hash-table-p <expr>) IS THIS A HASH TABLE?
Defined in common2.lsp
<expr> the expression to check
returns t if the value is a hash table, NIL otherwise

(keywordp <expr>) IS THIS A KEYWORD?
Defined in common2.lsp
<expr> the expression to check
returns t if the value is a keyword symbol, NIL otherwise

(packagep <expr>) IS THIS A PACKAGE?
Defined in common2.lsp
<expr> the expression to check
returns t if the value is a package, NIL otherwise

(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 <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 <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 <expr>) IS THIS NUMBER NEGATIVE?
<expr> the number to test
returns t if the number is negative, NIL otherwise

(zerop <expr>) IS THIS NUMBER ZERO?
<expr> the number to test
returns t if the number is zero, NIL otherwise

(plusp <expr>) IS THIS NUMBER POSITIVE?
<expr> the number to test
returns t if the number is positive, NIL otherwise

(evenp <expr>) IS THIS INTEGER EVEN?

| | |
|---|---|
| <expr> | the integer to test |
| returns | t if the integer is even, NIL otherwise |

(oddp <expr>) IS THIS INTEGER ODD?

| | |
|---|---|
| <expr> | the integer to test |
| returns | t if the integer is odd, NIL otherwise |

(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 <expr1> <expr2>) ARE THE EXPRESSIONS EQUAL?
(eql <expr1> <expr2>)
(equal <expr1> <expr2>)
(equalp <expr1> <expr2>)

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":<br>eq: identical pointers -- works with characters, symbols, and arbitrarily small integers<br>eql: works with all numbers, if same type (see also = on page 58)<br>equal: lists and strings<br>equalp: case insensitive characters (and strings), numbers of differing types, arrays (which can be equalp to string containing same elements) |

(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 (on page 97) or one of:

| | |
|---|---|
| ATOM | any atom |
| NULL | NIL |
| LIST | matches NIL or any cons cell |
| STREAM | any stream |
| NUMBER | any numeric type |
| REAL | flonum or rational number |
| INTEGER | fixnum or bignum |
| RATIONAL | fixnum or ratio |
| STRUCT | any structure (except hash-table) |
| FUNCTION | any function, as defined by functionp (page 73) |

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 <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 <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 <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 <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 <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 <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 <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

(typecase <expr> <case>...[(t <expr>)]) SELECT BY TYPE

macro defined in common2.lsp
<expr> the selection expression
<case> pair consisting of:
(<type> <expr>...)
where:
<type> type specifier as in function TYPEP (page 75)
<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 (<binding>...) <expr>...) CREATE LOCAL BINDINGS
(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 (<binding>...) <expr>...) CREATE LOCAL FUNCTIONS
(labels (<binding>...) <expr>...) FLET WITH RECURSIVE FUNCTIONS
(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 <sym> <expr>...) EVALUATE EXPRESSIONS AND CATCH THROWS

fsubr
<sym> the catch tag
<expr> expressions to evaluate
returns the value of the last expression or the throw expression

(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 <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 <expr>...) BASIC LOOPING FORM

fsubr

| | |
|---|---|
| <expr> | the body of the loop |
| returns | never returns (must use non-local exit, such as RETURN) |

(do (<binding>...) (<texpr> <rexpr>...) <expr>...) GENERAL LOOPING FORM
(do* (<binding>...) (<texpr> <rexpr>...) <expr>...)

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 (<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 (<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 (a fixnum) |
| <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 (<binding>...) <expr>...) THE PROGRAM FEATURE
(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 <name> <expr>...) NAMED BLOCK

fsubr
<name> the block name (quoted symbol)
<expr> the block body
returns the value of the last expression

(return [<expr>]) CAUSE A PROG CONSTRUCT TO RETURN A VALUE

fsubr
<expr> the value (defaults to NIL)
returns never returns

(return-from <name> [<value>]) RETURN FROM A NAMED BLOCK OR FUNCTION

fsubr. In traditional Xlisp, the names are dynamically scoped. A compilation option (default) uses lexical scoping like Common Lisp.
<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 <expr>...) BLOCK WITH LABELS

fsubr
<expr> expression(s) to evaluate or tags (symbols)
returns NIL

(go <sym>) GO TO A TAG WITHIN A TAGBODY

fsubr. In traditional Xlisp, tags are dynamically scoped. A compilation option (default) uses lexical scoping like Common Lisp.
<sym> the tag (quoted)
returns never returns

(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 <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 <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 <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 [<stream> [<eofp> [<eof> [<rflag>]]]]) READ AN EXPRESSION

NOTE: there has been an incompatible change in arguments from prior versions.

| | |
|---|---|
| <stream> | the input stream (default, or NIL, is *standard-input*, T is *terminal-io*) |
| <eofp> | When T, signal an error on end of file, when NIL return <eof> (default is T) |
| <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 <ch> <fcn> [ T ]) MODIFY READ TABLE

defined in init.lsp

| | |
|---|---|
| <ch> | character to define |
| <fcn> | function to bind to character (see page 12) |
| T | if TMACRO rather than NMACRO |

(get-macro-character <ch>) EXAMINE READ TABLE

defined in init.lsp

| | |
|---|---|
| <ch> | character |
| returns | function bound to character |

(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 <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, *print-base* for rationals, *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 <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 <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 [<stream>]) TERMINATE THE CURRENT PRINT LINE

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

(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 list was started, NIL if already at the start of a line. |

(flatsize <expr>) LENGTH OF PRINTED REPRESENTATION USING PRIN1

| | |
|---|---|
| <expr> | the expression |
| returns | the length |

(flatc <expr>) LENGTH OF PRINTED REPRESENTATION USING PRINC

| | |
|---|---|
| <expr> | the expression |
| returns | the length |

(y-or-n-p [<fmt> [<arg>...]]) ASK A YES OR NO QUESTION
(yes-or-no-p [<fmt> [<arg>...]])

defined in common.lsp. Uses *terminal-io* stream for interaction. y-or-n-p strives for a single character answer, using get-key if defined.

| | |
|---|---|
| <fmt> | optional format string for question (see page 84) |
| <arg> | arguments, if any, for format string |
| returns | T for yes, NIL for no. |

(prin1-to-string <expr>) PRINT TO A STRING
(princ-to-string <expr>)

defined in common2.lsp. Uses prin1 or princ conventions, respectively.

| | |
|---|---|
| <expr> | the expression to print |
| returns | the string containing the "printed" expression |

(read-from-string <str> [<eofp> [<eof>]] &key :start :end)
READ AN EXPRESSION

defined in common2.lsp.

| | |
|---|---|
| <str> | the input string |
| <eofp> | When T, signal an error on end of string, when NIL return <eof> (default is T) |
| <eof> | the value to return on end of string (default is NIL) |
| :start | starting index of <str>, default 0 |
| :end | ending index of <str>, default NIL (end of string) |
| returns | two values: the expression read and index of character after last one used |

# THE FORMAT FUNCTION

(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:

| | |
|---|---|
| ~? | use next argument as recursive format string |
| ~( ~) | process format string with case conversion |
| ~{ ~} | process format string repetitively |
| ~* | skip arguments |
| ~% | start a new line |
| ~& | start a new line if not on a new line |
| ~\n | ignore return and following whitespace |
| ~\| | start a new page |
| ~~ | print a tilde character |
| ~A or ~a | print next argument using princ |
| ~B or ~b | print next argument as binary integer (bignum extension) |
| ~D or ~d | print next argument as decimal 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 |
| ~O or ~o | print next argument as octal integer |
| ~R or ~r | print next number in any radix (bignum extension) |
| ~S or ~s | print next argument using prin1 |
| ~T or ~t | go to a specified column |
| ~X or ~x | print next argument as hexidecimal integer |
| ~[ ~; ~] | process format string conditionally |

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, the character '#' which represents the remaining number of arguments, 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, ~B, ~O, and ~X the full form is ("D" shown):

~mincol,padchar@D

If the data to print is not an integer, 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 ~R, the full form is:

~radix,mincol,padchar@R

The radix must be in the range 2 through 36. Other arguments are as in ~D, above. Unlike Common Lisp, English text and Roman numeral output is not supported.

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%, ~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 ~?, the next argument is taken as a format string, upon completion execution resumes in the current format string. The argument after is taken as the list of arguments used for the new format string unless the @ modifier is used, in which case the current argument list is used.

For ~(, the full form is ~(string~). The string is processed as a format string, however case conversion is performed on the output. If no modifiers are used, the string is converted to lowercase. If the colon modifier is used alone then all words are capitalized. If the @ modifier is used alone then the first character is converted to upper case and all remaining to lowercase. If both modifiers are used, all characters are converted to uppercase.

For ~{, the full form is ~n{string~}. Repeatedly processes string as a format string, or if the string is zero length, takes the next argument as the string. Iteration stops when processing has occured n times or no arguments remain. If the colon modifier is used on the ~} command, and n is non-zero then the string will be processed at least once. If no modifiers are used on ~{, then the arguments are taken from the next argument (like in ~?). If the colon modifier is used, the arguments are taken from the next argument which must be a list of sublists -- the sublists are used in turn to provide arguments on each iteration. In either case, the @ modifier will cause the current arguement list to be used rather than a single list argument.

For ~[, there are three formats. The first form is ~n[clause0~;clause1...~;clausen~]. Only one clause string is used, depending on the value of n. When n is absent, its value is taken from the argument list (as though 'v' had been used.) The last clause is treated as an "otherwise" clause if a colon modifier is used in its leading ~; command. The second form is ~:[clausenil~;clauset~]. The next argument is examined (and also consumed), and if nil clausenil is used, otherwise clauset is used. The third form is ~@[string~]. If then next argument is non-nil, then it is not used up and the format string is used, otherwise the argument is used up and the string is not used.

For ~*, the full form is ~n*. The count, n, defaults to 1 and is the number of arguments to skip. If the colon modifier is used, n is negated and skipping is backwards. The @ modifier causes n to be an absolute argument position (with default of 0), where the first argument is argument 0. Attempts to position before the first argument will position at the first argument, while attempts to position after the last argument signals an error.

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 (either -t on the command line or the 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 [<stream>[<eofp>[<eof>]]]) READ A CHARACTER FROM A STREAM

NOTE: New eof arguments are incompatible with older XLISP versions.

| | |
|---|---|
| <stream> | the input stream (default, or NIL, is *standard-input*, T is *terminal-io*) |
| <eofp> | When T, signal an error on end of file, when NIL return <eof> (default is T) |
| <eof> | the value to return on end of file (default is NIL) |
| returns | the character or <eof> at end of file |

(peek-char [<flag> [<stream> [<eofp> [<eof>]]]])

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*) |
| <eofp> | When T, signal an error on end of file, when NIL return <eof> (default is T) |
| <eof> | the value to return on end of file (default is NIL) |
| returns | the character or <eof> at end of file |

(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 [<stream>[<eofp>[<eof>]]]) READ A LINE FROM A STREAM

NOTE: New eof arguments are incompatible with older XLISP versions.

| | |
|---|---|
| <stream> | the input stream (default, or NIL, is *standard-input*, T is *terminal-io*) |
| <eofp> | When T, signal an error on end of file, when NIL return <eof> (default is T) |
| <eof> | the value to return on end of file (default is NIL) |
| returns | the string excluding the #\newline, or <eof> at end of file |

(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 (on page 97), or UNSIGNED-BYTE, SIGNED-BYTE, (UNSIGNED-BYTE <size>), or (SIGNED-BYTE <size>) with the bignum extension. CHARACTER (the default) is for text files, the other types are for binary files and can only be used with READ-BYTE and WRITE-BYTE. FIXNUM is a vestige of older XLISP-PLUS releases and is identical to (UNSIGNED-BYTE 8). If no size is given, then size defaults to 8. Size must be a multiple of 8. |
| :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 <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. |

(probe-file <fname>) CHECK FOR EXISTANCE OF A FILE

Defined in common2.lsp

| | |
|---|---|
| <fname> | file name string or symbol |
| returns | t if file exists, else NIL |

(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 <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 (<var> <fname> [<karg>...]) [<expr>...])
EVALUATE USING A FILE

(with-open-stream (<var> <stream>) [<expr>...])
EVALUATE USING AN OPENED STREAM

Defined in common.lsp and common2.lsp, respectively, as macros. Stream will always be closed upon completion

| | |
|---|---|
| <var> | symbol name to bind stream to while evaluating expresssions (quoted) |
| <fname> | file name string or symbol |
| <stream> | a file or string stream |
| <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 <stream>[<eofp>[<eof>]]) READ A BYTE FROM A STREAM

NOTE: New eof arguments are incompatible with older XLISP versions. Stream argument used to be optional. Number of system bytes read depend on :element-type specified in the open command.

| | |
|---|---|
| <stream> | the input stream |
| <eofp> | When T, signal an error on end of file, when NIL return <eof> (default is T) |
| <eof> | the value to return on end of file (default is NIL) |
| returns | the byte (integer) or <eof> at end of file |

(write-byte <byte> <stream>) WRITE A BYTE TO A STREAM

NOTE: Stream argument used to be optional. Number of system bytes writen depend on :element-type specified in open command. No checks are made for overflow, however negative values cannot be written to unsigned-byte streams.

| | |
|---|---|
| <byte> | the byte to write (integer) |
| <stream> | the output stream |
| returns | the byte (integer) |

(file-length <stream>) GET LENGTH OF FILE

For a CHARACTER 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 <stream> [<expr>]) GET OR SET FILE POSITION

For a CHARACTER 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' string and list return a sting 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 <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)

| | |
|---|---|
| returns | an unnamed output stream |

(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 <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 (<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 (<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 [<sym>...]) — ADD A FUNCTION TO THE TRACE LIST

fsubr
<sym> the function(s) to add (quoted)
returns the trace list

(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 <emsg> {<arg>}) — SIGNAL A NON-CORRECTABLE ERROR

Note that the definition of this function has changed from 2.1e and earlier so to match Common Lisp.
<emsg> the error message string, which is processed by FORMAT
<arg> optional argument{s} for FORMAT
returns never returns

(cerror <cmsg> <emsg> {<arg>}) — SIGNAL A CORRECTABLE ERROR

Note that the definition of this function has changed from 2.1e and earlier so to match Common Lisp.
<cmsg> the continue message string, which is processed by FORMAT
<emsg> the error message string, which is processed by FORMAT
<arg> optional argument(s) for both FORMATs (arguments are useable twice)
returns NIL when continued from the break loop

(break <bmsg> {<arg>}) — ENTER A BREAK LOOP

Note that the definition of this function has changed from 2.1e and earlier so to match Common Lisp.
<bmsg> the break message string, which is processed by FORMAT
<arg> optional argument(s) for FORMAT
returns NIL when continued from the break loop

(clean-up) — CLEAN-UP AFTER AN ERROR

returns never returns

(top-level) — CLEAN-UP AFTER AN ERROR AND RETURN TO THE TOP LEVEL

Runs the function in variable *top-level-loop* (ususally TOP-LEVEL-LOOP)
returns never returns

(continue) — CONTINUE FROM A CORRECTABLE ERROR

returns never returns

(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 [<n>]) — PRINT N LEVELS OF TRACE BACK INFORMATION

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

(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 <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) ENABLE DEBUG BREAKS
(nodebug) DISABLE DEBUG BREAKS

Defined in init.lsp

(ecase <expr> <case>...) SELECT BY CASE
(ccase <expr> <case>...)

Defined in common2.lsp as macros. ECASE signals a non-continuable error if there are no case matches, while CCASE signals a continuable error and allows changing the value of <expr>.

<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
returns the value of the last expression of the matching case

(etypecase <expr> <case>...) SELECT BY TYPE
(ctypecase <expr> <case>...)

Defined in common2.lsp as macros. ETYPECASE signals a non-continuable error if there are no case matches, while CTYPECASE signals a continuable error and allows changing the value of <expr>.

<expr> the selection expression
<case> pair consisting of:
(<type> <expr>...)
where:
<type> type specifier as in function TYPEP (page 75)
<expr> are expressions to execute if the case matches
returns the value of the last expression of the matching case

(check-type <place> <type> [<string>]) VERIFY DATA TYPE

Defined in common2.lsp as a macro. If value stored at <place> is not of type <type> then a continuable error is signaled which allows changing the value at <place>.

| | |
|---|---|
| <place> | a valid field specifier (generalized variable) |
| <type> | a valid type specifier as in function TYPEP (page 75) |
| <string> | string to print as the error message |
| returns | NIL |

(assert <test> [([<place>...]) [<string> [<args>...]]]) MAKE AN ASSERTION

Defined in common2.lsp. If value of <test> is NIL then a continuable error is signaled which allows changing the place values.

| | |
|---|---|
| <test> | assertion test |
| <place> | zero or more valid field specifiers |
| <string> | error message printed using FORMAT (evaluated only if assertion fails) |
| <args> | arguments for FORMAT (evaluated only if assertion fails) |
| returns | NIL |

## SYSTEM FUNCTIONS

(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 <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 <fname>) SAVE WORKSPACE TO A FILE

You cannot save from within a load. Not all of the state may be saved -- see "restore", above. 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 <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 [<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) FORCE GARBAGE COLLECTION

| | |
|---|---|
| returns | NIL |

(expand [<num>]) EXPAND MEMORY BY ADDING SEGMENTS

| | |
|---|---|
| <num> | the (fixnum) number of segments to add, default 1 |
| returns | the (fixnum) number of segments added |

(alloc <num> [<num2> [<num3>]]) CHANGE SEGMENT SIZE

| | |
|---|---|
| <num> | the (fixnum) number of nodes to allocate |
| <num2> | the (fixnum) number of pointer elements to allocate in an array segment (when dynamic array allocation compiled). Default is no change. |
| <num3> | the <fixnum> ideal ratio of free to used vector space (versions of XLISP using dldmem.c). Default is 1. Increase if extensive time is spent in garbage collection in bignum math intensive programs. |
| returns | the old number of nodes to allocate |

(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
Time - time spent performing garbage collections (in seconds)

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

Vector nodes - total vector space (pointers and string, in pointer sized units)
Vector free - free space in vector area (may be fragmented across segments)
Vector segs - number of vector segments. Increases and decreases as needed.
Vec allocate - number of pointer elements to allocate in any new vector segment
Vec collect - number of garbage collections instigated by vector space exhaustion

returns NIL

(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* |

(sleep <expr>) TIME DELAY

defined in common2.lsp

| | |
|---|---|
| <expr> | time in seconds |
| returns | NIL, after <expr> seconds delay |

(get-internal-real-time) GET ELAPSED CLOCK TIME
(get-internal-run-time) GET ELAPSED EXECUTION TIME

| | |
|---|---|
| returns | integer time in system units (see internal-time-units-per-second on page 24). meaning of absolute values is system dependent. |

(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, integers can be coerced into characters or flonums. Ratios can be coerced into flonums. Flonums can be coerced into complex.

| | |
|---|---|
| <expr> | the expression to coerce |
| <type> | desired type, as returned by type-of (see page 97) |
| returns | <expr> if type is correct, or converted object. |

(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 |
| | BIGNUM | for large 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 <addrs>) PEEK AT A LOCATION IN MEMORY

| | |
|---|---|
| <addrs> | the address to peek at (fixnum) |
| returns | the value at the specified address (integer) |

(poke <addrs> <value>) POKE A VALUE INTO MEMORY

| | |
|---|---|
| <addrs> | the address to poke (fixnum) |
| <value> | the value to poke into the address (fixnum) |
| returns | the value |

(address-of <expr>) GET THE ADDRESS OF AN XLISP NODE

| | |
|---|---|
| <expr> | the node |
| returns | the address of the node (fixnum) |

(get-key) READ A KEYSTROKE FROM CONSOLE

OS dependent.

| | |
|---|---|
| returns | integer value of key (no echo) |

(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) |

(set-stack-mark <size>) SET SYSTEM STACK WARNING POINT

OS dependent -- not always available. The system will perform a continuable error when the amount of remaining system stack passes below this setting. The trap is reset at the top-level. This function is useful for debugging runaway recursive functions.

| | |
|---|---|
| <size> | Remaining stack, in bytes. Minimum value is fixed at the value that causes the system stack overflow error, while the maximum value is limitied to somewhat less than the current remaining stack space. Use "0" to turn the warnings off. |
| returns | the previous value. |

(top-level-loop) DEFAULT TOP LEVEL LOOP

Runs the XLISP top level read-eval-print loop, described earlier. Never returns.

(reset-system) FLUSH INPUT BUFFERS

Used by user-implemented top level loops to flush the input buffer

| | |
|---|---|
| returns | NIL |

(exit) EXIT XLISP

| | |
|---|---|
| returns | never returns |

(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 |
| BIGNUM | copy as a string |
| structure | copy as an ARRAY |

(eval-when <condition> <body> ...)

Macro defined in common.lsp, and provided to assist in porting Common Lisp applications to XLISP-PLUS.

| | |
|---|---|
| <condition> | List of conditions |
| <body> | expressions which are evaluated if one of the conditions is EXECUTE or LOAD. |
| returns | result of last body expression |

The following graphic and display functions represent an extension by Tom Almy:

(cls) CLEAR DISPLAY

Clear the display and position cursor at upper left corner.

returns nil

(cleol) CLEAR TO END OF LINE

Clears current line to end.

returns nil

(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

(mode <ax> [<bx> <width> <height>) SET DISPLAY MODE

Standard modes require only <ax> argument. Extended modes are "Super-VGA" or "Super-EGA" and are display card specific. Not all XLISP versions support all modes.

<ax> Graphic mode (value passed in register AX)
Common standard Modes:
0,1 - 40x25 text
2,3 - 80x25 text
4,5 - 320x200 4 color graphics (CGA)
6 - 640x200 monchrome graphics (CGA)
13 - 320x200 16 color graphics (EGA)
14 - 640x200 16 color graphics (EGA)
16 - 640x350 16 color graphics (EGA)
18 - 640x480 16 color graphics (VGA)
19 - 320x200 256 color graphics (VGA)

<bx> BX value for some extended graphic modes

<width> width for extended graphic modes

<height> height for extended graphic modes

returns a list of the number of columns, number of lines (1 for CGA), maximum X graphic coordinate (-1 for text modes), and the maximum Y graphic coordinate (-1 for text modes), or NIL if fails

(color <value>) SET DRAWING COLOR

<value> Drawing color (not checked for validity)

returns <value>

(move <x1> <y1> [<x2> <y2> ...]) ABSOLUTE MOVE

(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 [<x1> <y1> ...]) ABSOLUTE DRAW
(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 |

# ADDITIONAL FUNCTIONS AND UTILITIES

**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.

When the package feature is compiled in, the debugger is in the TOOLS package.

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.

When the package feature is compiled in, these funtions are in the TOOLS package.

(pp <object> [<stream>]) PRETTY PRINT EXPRESSION
(pp-def <funct> [<stream>]) PRETTY PRINT FUNCTION/MACRO
(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.

**DOCUMENT.LSP**

This file provides the documentation feature of Common Lisp. When loaded, glossary descriptions of system functions and variables are installed from the file GLOS.TXT. References are made directly to the file so that the size of the XLISP image will not increase. The following functions are implemented:

(documentation <symbol> <doctype>) GET DOCUMENTATION STRING

Use with SETF to alter documentation string.

| | |
|---|---|
| <symbol> | Symbol of interest |
| <doctype> | Documentation type, one of FUNCTION, VARIABLE, STRUCTURE, SETF, or TYPE. |
| returns | Documentation string |

(glos <symbol> [T]) GET DOCUMENTATION

Defined in package TOOLS.

| | |
|---|---|
| <symbol> | Either the symbol for which the documentation is requested, or a string which will match all symbol names containing that string. |
| T | Flag saying to treat symbol as a string, and match all related names. |
| returns | nothing |

Documentation can be added via the DEFCONSTANT, DEFPARAMETER, DEFVAR, DEFUN, DEFMACRO, and DEFSTRUCT functions as well as via DOCUMENTATION. Documentation is stored in the property list in properties %DOC-FUNCTION, %DOC-STRUCTURE, %DOC-VARIABLE, %DOC-SETF, and %DOC-TYPE. The latter two are not currently used. These properties either contain the documentation string or the offset into the GLOS.TXT file.

**INSPECT.LSP**

INSPECT.LSP contains an XLISP editor/inspector. When the package feature is compiled in, the editor is in the TOOLS package. Two functions, INSPECT and DESCRIBE, are part of Common Lisp and are in the XLISP package.

(ins <symbol>) INSPECT A SYMBOL
(inspect <expr>) INSPECTOR
(insf <symbol>) INSPECT FUNCTION BINDING

INS and INSF are macros defined in package TOOLS. INSF edits the function binding and allows changing the argument list or type (MACRO or LAMBDA).

| | |
|---|---|
| <symbol> | Symbol to inspect (quoted) |
| <expr> | Expression to inspect |
| returns | Symbol or expression |

(describe <expr>) DESCRIBE

Tells what <expr> is, but doesn't allow editing. Use INSPECT to edit.

| | |
|---|---|
| <expr> | Expression to describe |
| returns | The expression |

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. Also, naturally, changing the values of structure elements, instance variables, or symbols cannot be undone.

For all commands taking a numeric argument, the first element of the selection is the 0th (as in NTH function).

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, the class variables, 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). Structure elements can be examined and changed.

(command list on next page)

COMMANDS (all "hot keyed and case sensitive"):

| | |
|---|---|
| ? | List available commands |
| A | select the CAR of the current selection. |
| D | select the CDR of the current selection. |
| e n | select ("Edit") element n |
| r n x | Replaces element n with x. |
| X | eXit, saving all changes |
| Q | Quit, without saving changes |
| b | go Back one level (backs up A, D or e commands) |
| B n | go Back n levels. |
| l | List selection using pprint; if selection is symbol, give short description |
| v | Verbosity toggle |
| . n | change maximum print length (default 10) |
| # n | change maximum print depth (default 3) |
| ! x | evaluates x and prints result, the symbol tools:@ is bound to the selection |
| R x | Replaces the selection with evaluated x, the symbol tools:@ is bound to the selection |

ADDITIONAL COMMANDS (selection is a list or array):

| | |
|---|---|
| ( n m | inserts parenthesis starting with the nth element, for m elements. |
| ) n | removes parenthesis surrounding nth element of selection, which may be array or list |
| [ n m | as in (, but makes elements into an array |
| i n x | Inserts x before nth element in selection. |
| d n | Deletes nth element in selection. |
| S x y | Substitute all occurances of y with x in selection (which must be a list). EQUAL is used for the comparison. |

# COMPILATION OPTIONS

XLISP PLUS has many compilation options to optimize the executable for specific tasks. These are the most useful:

1. Available Functions (all turned on by default)
   - SRCHFCN supplies SEARCH
   - MAPFCNS supplies SOME EVERY NOTANY NOTEVERY and MAP
   - POSFCNS supplies POSITION-* COUNT-* and FIND-* functions
   - REMDUPS supplies REMOVE-DUPLICATES
   - REDUCE supplies REDUCE
   - SUBSTITUTE supplies SUBSTITUTE-* and NSUBSTITUTE-*
   - ADDEDTAA supplies GENERIC
   - TIMES supplies TIME GET-INTERNAL-RUN-TIME GET-INTERNAL-REAL-TIME and the constant INTERNAL-TIME-UNITS-PER-SECOND
   - RANDOM supplies RANDOM-NUMBER-STATE type, *RANDOM-STATE*, and the function MAKE-RANDOM-STATE. Requires TIMES.
   - HASHFCNS supplies SETHASH MAKE-HASH-TABLE REMHASH MAPHASH CLRHASH and HASH-TABLE-COUNT
   - SETS supplies ADJOIN UNION INTERSECTION SET-DIFFERENCE and SUBSETP
   - SAVERESTORE supplies SAVE and RESTORE
   - GRAPHICS supplies graphic functions (when available)
2. Features (all turned on by default)
   - COMPLX adds complex number support including math functions COMPLEX COMPLEXP IMAGPART REALPART CONJUGATE PHASE LOG FLOOR CEILING ROUND PI LCM and ASH
   - BIGNUMS adds bignum, ratio, and read/print radix support. Requires COMPLX.
   - NOOVFIXNUM Check for fixnum overflow, and convert to flonum (only applies if BIGNUMS not used)
   - PACKAGES uses the packages implementation. Some people find XLISP PLUS easier to use if this is not defined.
   - MULVALS multiple value returns
   - FILETABLE files referenced via a table -- allows saving and restoring open files (in WKS files), and is required by Microsoft Windows versions. Also allows functions TRUENAME and DELETE-FILE.
   - KEYARG adds :key keyword option to functions having it
   - FROMEND adds the :from-end and :count keywords to functions having them
   - AOKEY makes &allow-other-keys functional. Without this option, all functions behave as though &allow-other-keys is always specified.
   - APPLYHOOK adds applyhook support
3. Backwards compatibility
   - OLDERRORS makes CERROR and ERROR work as in XLISP-PLUS 2.1e or earlier, which is not compatible with more recent versions (or Common Lisp)
   - LEXBIND lexical tag scoping for TAGBODY/GO and BLOCK/RETURN, as in Common Lisp. If not defined, then the original dynamic scoping is used.
4. Environmental options
   - ASCII8 eight bit ASCII character support
   - ANSI8 used in addition to ASCII8 for MS Windows character code page

- READTABLECASE adds *readtable-case* and its functionality
- PATHNAMES allows environment variable to specify search path for RESTORE and LOAD functions
- BIGENDIANFILE binary files use "bigendian" orientation. Normally this option is defined when BIGENDIAN (required on bigendian systems) is defined, but this has been made a separate option to allow file portability between systems.

5. Performance options
   - JMAC increases performance slightly, except on 16 bit DOS environments.
   - GENERIC use generic bignum to float conversion. Required for bigendian or non IEEE floating point systems (such as Macs). Using this selection decreases precision and increases execution time.

In addition, there are options for various stack sizes, static fixnum values, and various allocation sizes that can be altered. They have been set optimally for each compiler/environment and "typical" applications.

# BUG FIXES AND EXTENSIONS

In this section, CL means "Common Lisp compatible to the extent possible". CX means "now works with complex numbers". CR means "now works with ratios". * means "implemented in LISP rather than C". # means "implementation moved from LISP to C".

### Bug Fixes

RESTORE did not work -- several bugs for 80x86 systems. Only one restore would work per session -- all systems.

:downcase for variable *printcase* did not work with some compilers.

Modifications to make the source acceptable to ANSI C compilers.

Values for ADEPTH and EDEPTH changed to more reasonable values -- before this change the processor stack would overflow first, causing a crash.

On systems with 16 bit integers: STRCAT crashes when aggregate size of argument strings were greater than 32k. MAKE-ARRAY crashes on too-large arrays. DOTIMES, AREF, AREF and NTH place forms of SETF, MAKE-STRING-INPUT-STREAM and GET-OUTPUT-STREAM-STRING treat numeric argument modulo 65536. MAKE-STRING-INPUT-STREAM did not check for start>end.

Strings containing nulls could not be read or printed.

NTH and NTHCDR failed for zero length lists.

Unnamed streams did not survive garbage collections.

(format nil ...) did not protect from garbage collection the unnamed stream it creates.

SORT did not protect some pointers from garbage collection.

SYMBOL-NAME SYMBOL-VALUE SYMBOL-PLIST BOUNDP and FBOUNDP failed with symbol NIL as argument.

LAST returned wrong value when its argument list ended with a dotted pair.

*gc-hook* was not rebound to NIL during execution of gchook function, causing potential infinite recursion and crash.

Executing RETURN from within a DOLIST or DOTIMES caused the environment to be wrong.

When errors occured during loading, which were not caught, the file would be left open. EVAL and LOAD did not use global environment. EVALHOOK's default environment was not global.

Invalid symbols (those containing control characters, for instance), can no longer be created with intern and make-symbol.

The key T, meaning "otherwise" in the CASE function used to be allowed in any position. Now it only means "otherwise" when used as the last case.

The lexical and functional environment of send of :answer (which defines a new method) are now used during the method's evaluation, rather than the global environment.

Signatures added for WKS files so that invalid ones will be rejected.

Checks added for file names and identifier names being too long.

Indexing code fixed to allow almost 64k long strings in 16 bit systems. It is no longer possible to allocate arrays or strings that are too long for the underlying system.

Circularity checks added to PRINT LAST BUTLAST LENGTH MEMBER and MAP functions. An error is produced for all but MEMBER, which will execute correctly.

Code for SETF modified so that a Common Lisp compatible DEFSETF could be used.

Circularity checks added to EQUAL.

Check for even number of arguments to SETQ, SETF, and PSETQ added. PSETQ changed to return NIL rather than result of first assignment (really now!).

## User Interface Changes

-w command line argument to specify alternate or no workspace.

-b command line argument for batch operation.

-? command line argument gives usage message.

init.lsp not loaded if workspace loaded.

Search path can be provided for workspaces and .lsp files.

Standard input and output can be redirected. *TERMINAL-IO* stream added which is always bound to console (stderr).

Non-error messages are sent to *DEBUG-IO* so they don't clutter *STANDARD-OUTPUT*

Results of evaluations are printed on a fresh line rather than at the end of the preceeding line (if any). This enhances readability.

Display writes are buffered.

Character literals available for all 256 values. CL

Uninterned symbols print with leading #:. CL

PRIN1 generates appropriate escape sequences for control and meta characters in strings. CL

Read macro #. added. CL

Lisp code for nested backquote macros added. CL

Read macro #C added for complex numbers. CL

Semantics for #S read macro changed so that it can read in structures written by PRINT. CL

PRINT of file streams shows file name, or "closed" if a closed file stream.

*PRINT-CASE* now applies to PRINC. CL

Added *READTABLE-CASE* to control case conversion on input and output, allowing case sensitive code. CL-like

Reader macros #+ and #- added, along with global variable *FEATURES*. CL

Added optional and OS dependent checking of system stack overflow, with checks in READ, PRINT, EVAL, and in the garbage collector. Added a new function SET-STACK-MARK which performs a continuable error when the remaining stack space drops below a preset amount.

Improved command line editing, symbol name lookup, and history (command recall) for MS-DOS.

*PRINT-CASE* can now be :CAPITALIZE. CL

Packages added.

Reader macro #nR added. Added *READ-BASE* to control default read radix. Numeric input is now Common Lisp compliant.

## New/Changed Data Types

NIL -- was treated as a special case, now just a normal symbol.
symbols -- value binding can optionally be constant or special. "*unbound*" is no longer a symbol so does not have to be specially treated.
ratio numbers -- new type.
complex numbers -- new type, can be rational or real. (Older versions allowed fixnum or real.)
bignums -- new type.
character strings -- The ASCII NUL (code 0) is now a valid character.
objects -- objects of class Class have a new instance variable which is the print name of the class.
hash-table -- new type, close to CL
random-state -- new type, CL
Property list properties are no longer limited to just symbols CL
Multiple value returns added where appropriate
Packages added where appropriate

## New Variables and Constants

*apply-hook* Now activated
*command-line*

*displace-macros* Macros are replaced with their expansions when possible *dos-input* MSDOS only, uses DOS interface to interact with user. Allows recall of earlier command(s).
*load-file-arguments*
*print-level* CL
*print-length* CL
*random-state* CL
*ratio-format*
*readtable-case* CL-like
*startup-functions*
*terminal-io* CL
*top-level-loop*
internal-time-units-per-second CL
pi CL
*read-base*
*print-base*

**New functions**

ACONS CL*
ACOSH CL*
ADJOIN CL
ALPHA-CHAR-P CL
APPLYHOOK CL
APROPOS CL*
APROPOS-LIST CL*
ASH CL
ASINH CL*
ATANH CL*
BUTLAST CL
CEILING CL
CIS CL*
CLREOL (clear to end of line -- MS/DOS only)
CLRHASH CL
CLS (clear screen -- MS/DOS only)
COERCE CL
COLOR (graphics -- MS/DOS only)
COMPLEMENT CL
COMPLEX CL
COMPLEXP CL
CONCATENATE CL
CONJUGATE CL
CONSTANTP CL
COPY-ALIST CL*
COPY-LIST CL*
COPY-TREE CL*
COSH CL*
COUNT-IF CL
COUNT-IF-NOT CL
DECF CL*
DECLARE *
DEFCLASS * (define a new class)

DEFINST * (define a new instance)
DEFMETHOD * (define a new method)
DEFPACKAGE CL*
DEFSETF CL*
DELETE-FILE CL
DELETE-PACKAGE CL
DENOMINATOR CL
DESCRIBE CL*
DO-ALL-SYMBOLS CL*
DO-EXTERNAL-SYMBOLS CL*
DO-SYMBOLS CL*
DOCUMENTATION CL*
DRAW (graphics -- MS/DOS only)
DRAWREL (graphics -- MS/DOS only)
ELT CL
EQUALP CL*
EVAL-WHEN *
EVERY CL
EXPORT CL
FILE-LENGTH CL
FILE-POSITION CL
FILL CL*
FIND-ALL-SYMBOLS CL
FIND-IF CL
FIND-IF-NOT CL
FIND-PACKAGE CL
FLOOR CL
FRESH-LINE CL
FUNCTIONP CL*
GENERIC (implementation debugging function)
GET-INTERNAL-REAL-TIME CL
GET-INTERNAL-RUN-TIME CL
GETF CL
GETHASH CL
GOTO-XY (position cursor -- MS/DOS only)
HASH-TABLE-COUNT CL
IDENTITY CL
IMAGPART CL
IMPORT CL
INCF CL*
IN-PACKAGE CL
INPUT-STREAM-P CL
INSPECT CL*
INTEGER-LENGTH CL
INTERSECTION CL
LCM CL
LIST* CL
LIST-ALL-PACKAGES CL
LIST-LENGTH CL
LOG CL
LOGANDC1 CL

LOGANDC2 CL
LOGBITP CL
LOGCOUNT CL
LOGEQV CL
LOGNAND CL
LOGNOR CL
LOGORC1 CL
LOGORC2 CL
LOGTEST CL
MAKE-HASK-TABLE CL
MAKE-PACKAGE CL
MAKE-RANDOM-STATE CL
MAP CL
MAP-INTO CL
MAPHASH CL
MARK-AS-SPECIAL
MODE (graphics -- MS/DOS only)
MOVE (graphics -- MS/DOS only)
MOVEREL (graphics -- MS/DOS only)
MULTIPLE-VALUE-BIND CL*
MULTIPLE-VALUE-CALL CL
MULTIPLE-VALUE-LIST CL*
MULTIPLE-VALUE-PROG1 CL
MULTIPLE-VALUE-SETQ CL*
NINTERSECTION CL*
NTH-VALUE
NOTANY CL
NOTEVERY CL
NREVERSE CL
NSET-DIFFERENCE CL*
NSET-EXCLUSIVE-OR CL*
NSTRING-CAPITALIZE CL
NSUBSTITUTE CL
NSUBSTITUTE-IF CL
NSUBSTITUTE-IF-NOT CL
NUMERATOR CL
NUNION CL*
OPEN-STREAM-P CL
OUTPUT-STREAM-P CL
PACKAGE-NAME CL
PACKAGE-NICKNAMES CL
PACKAGE-OBARRAY
PACKAGE-SHADOWING-SYMBOLS CL
PACKAGE-USED-BY-LIST CL
PACKAGE-USE-LIST CL
PACKAGE-VALID-P
PAIRLIS CL*
PHASE CL
POP CL*
POSITION-IF CL
POSITION-IF-NOT CL

PROCLAIM *
PSETF CL
PUSH CL*
PUSHNEW CL*
RATIONAL CL
RATIONALP CL
REALPART CL
REDUCE CL except no :from-end
REMF CL*
REMHASH CL
REMOVE-DUPLICATES CL except no :from-end
RENAME-PACKAGE CL
REPLACE CL*
RESET-SYSTEM
ROUND CL
SEARCH CL except no :from-end
SET-DIFFERENCE CL
SET-EXCLUSIVE-OR CL*
SET-STACK-MARK
SETF Placeform ELT CL
SETF Placeform GETF CL
SETF Placeform GETHASH CL
SETF Placeform SEND* (set instance variable)
SHADOW CL
SHADOWING-IMPORT CL
SIGNUM CL*
SINH CL*
SOME CL
SPECIALP CL
STABLE-SORT CL
STRING-CAPITALIZE CL
SUBSETP CL
SUBSTITUTE CL
SUBSTITUTE-IF CL
SUBSTITUTE-IF-NOT CL
SYMBOL-PACKAGE CL
TANH CL*
TIME CL
TOP-LEVEL-LOOP
TRUENAME CL
TYPEP CL
UNEXPORT CL
UNINTERN CL*
UNION CL
UNUSE-PACKAGE CL
USE-PACKAGE CL
VALUES CL
VALUES-LIST CL
WITH-INPUT-FROM-STRING CL*
WITH-OPEN-FILE CL*
WITH-OUTPUT-TO-STRING CL*

Y-OR-N-P CL*
YES-OR-NO-P CL*

## Changed functions

&ALLOW-OTHER-KEYS CL (now functions, is no longer ignored)
:ALLOW-OTHER-KEYS CL
* CL CR CX (with no arguments, returns 1)
+ CL CR CX (with no arguments, returns 0)
- CL CR CX
/ CL CR CX
1+ CL CR CX
1- CL CR CX
ABS CL CR CX
ACOS CL CR CX
ALLOC (new optional second argument)
APPLY CL (allows multiple arguments)
AREF CL (now works on strings)
ASIN CL CR CX
ASSOC CL (added :key)
ATAN CL CR CX (second argument now allowed)
BREAK CL
CERROR CL
CHAR-CODE CL (parity bit is stripped)
CLOSE CL (will close unnamed stream strings)
COS CL CR CX
DEFCONSTANT CL# (true constants)
DEFPARAMETER CL# (true special variables)
DEFSTRUCT (added option :print-function, comment field)
DEFVAR CL# (true special variables)
DELETE (added keywords :key :start :end :count :from-end. Works on arrays and strings)
DELETE-IF (added keywords :key :start :end :count :from-end. Works on arrays and strings)
DELETE-IF-NOT (added keywords :key :start :end :count :from-end. Works on arrays and strings)
DIGIT-CHAR CL (added optional radix argument)
DIGIT-CHAR-P CL (added optional radix argument)
ERROR CL
EXP CL CR CX
EXPT CL CR CX
FMAKUNBOUND #
FORMAT (added directives # ~B ~D ~E ~F ~G ~O ~R ~X ~& ~* ~? ~| ~( ~[ ~{ ~T ~\N and lowercase directives)
GET CL
HASH (hashes everything, not just symbols or strings)
LOAD CL (uses path to find file, allows file stream for name argument)
LOGAND CL (with no arguments, returns -1)
LOGIOR CL (with no arguments, returns 0)
LOGXOR CL (with no arguments returns 0)
MAKE-ARRAY (added keywords :initial-contents and :initial-element)
MAKE-STRING-INPUT-STREAM CL (:end NIL means end of string)
MAKUNBOUND #

MAPCAN #
MAPCON #
MEMBER CL (added :key)
NSTRING-DOWNCASE CL (string argument can be symbol, :end NIL means end of string)
NSTRING-UPCASE CL (string argument can be symbol, :end NIL means end of string)
NSUBLIS CL
NSUBST CL
NSUBST-IF CL
NSUBST-IF-NOT CL
OPEN CL (many additional options, as in Common Lisp)
PEEK (fixnum sized location is fetched)
PEEK-CHAR CL (input stream NIL is *standard-input*, T is *terminal-io*, eof arguments)
POKE (fixnum sized location is stored)
PPRINT (output stream NIL is *standard-output*, T is *terminal-io*)
PRIN1 CL (output stream NIL is *standard-output*, T is *terminal-io*)
PRINC CL (output stream NIL is *standard-output*, T is *terminal-io*)
PRINT (output stream NIL is *standard-output*, T is *terminal-io*)
RANDOM CL (works with random-states)
READ (input stream NIL is *standard-input*, T is *terminal-io*, eof arguments)
READ-BYTE CL
READ-CHAR CL (input stream NIL is *standard-input*, T is *terminal-io*, eof arguments)
READ-LINE CL (input stream NIL is *standard-input*, T is *terminal-io*, eof arguments)
REM CR CL (only two arguments now allowed, may be floating point)
REMOVE (added keywords :key :start :end :count :from-end. Works on arrays and strings)
REMOVE-IF (added keywords :key :start :end :count :from-end. Works on arrays and strings)
REMOVE-IF-NOT (added keywords :key :start :end :count :from-end. Works on arrays and strings)
RESTORE (uses path to find file, restores file streams, fine name argument may be file stream)
REVERSE CL (works on arrays and strings)
ROUND CL (rounds to nearest even)
SAVE (file name argument may be file stream)
SIN CL CR CX
SORT (added :key) CL (with most compilers)
SQRT CL CR CX
STRCAT * (now a macro, use of CONCATENATE is recommended)
STRING-comparisonFunctions CL (string arguments can be symbols)
STRING-DOWNCASE CL (string argument can be symbol, :end NIL means end of string)
STRING-LEFT-TRIM CL (string argument can be symbol)
STRING-RIGHT-TRIM CL (string argument can be symbol)
STRING-TRIM CL (string argument can be symbol)
STRING-UPCASE CL (string argument can be symbol, :end NIL means end of string)
SUBLIS CL (modified to do minimum copying)
SUBSEQ CL (works on arrays and lists)
SUBST CL (modified to do minimum copying)
TAN CL CR CX
TERPRI CL (output stream NIL is *standard-output*, T is *terminal-io*)
TRUNCATE CR CL (allows denominator argument)
TYPE-OF (returns HASH-TABLE for hashtables, COMPLEX for complex, and LIST for NIL)
UNTRACE CL (with no arguments, untraces all functions)
VALUES CL
VALUES-LIST CL
WRITE-BYTE CL

WRITE-CHAR CL (output stream NIL is *standard-output*, T is *terminal-io*)

## New messages for class Object

:prin1 <stream>
:superclass *
:ismemberof <cls> *
:iskindof <cls> *
:respondsto <selector> *
:storeon (returns form that will create a copy of the object) *

## New messages for class Class

:superclass *
:messages *
:storeon (returns form that will recreate class and methods) *

# 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 give an error (or if a second nil argument is specified, will return nil or whatever value was supplied as the third 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 nil) (read fp nil)))
     ((null ex) (close fp) nil)
  (print ex))
```

The file will be closed with the next garbage collection.

# INDEX