XLISP-PLUS: Another Object-oriented Lisp

Version 2.1d

January 2, 1992

Tom Almy
toma@sail.labs.tek.com

Table of Contents

INTRODUCTION

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

Since XLISP-PLUS is based on XLISP, most XLISP programs will run on XLISP-PLUS. Since XLISP-PLUS incorporates many more features of Common Lisp, many small Common Lisp applications will run on XLISP-PLUS with little modification. See the section starting on page 85 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 in this document.

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

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 providing no workspace file was loaded, XLISP attempts to load "init.lsp" from the current directory. It then loads any files named as parameters on the command line (after appending ".lsp" to their names). If the "-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".

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

```
>
```

This indicates that XLISP is waiting for an expression to be typed.

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

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

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

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

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

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, and continue instead of ctrl-C, ctrl-G, and ctrl-P.

BREAK COMMAND LOOP

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

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

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

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

If the symbol '*breakenable*' is NIL, XLISP looks for a surrounding errset function. If one is found, XLISP examines the value of the print flag. If this flag is true, the error message is printed. In any case, XLISP causes the errset function call to return NIL.

If there is no surrounding errset function, XLISP prints the error message and returns to the top level.

DATA TYPES

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

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

- NIL
  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.
- ratios
  The CAR field is used to hold the numerator while the CDR field is used to hold the denominator. The numerator is a 32 bit signed value while the denominator is a 31 bit positive value.
- characters
  All characters are statically allocated and are thus EQ characters of the same value. The CAR field is used to hold the value. In XLISP characters are "unsigned" and thus range in value from 0 to 255.
- flonums (floating point numbers)
  The CAR and CDR fields hold the value, which is typically a 64 bit IEEE floating point number.

- complex numbers
  Part of the math extension compilation option. Internally implemented as an array of the real and imaginary parts. The parts can be either both fixnums or both flonums. Any function which would return an fixnum complex number with a zero imaginary part returns just the fixnum.
- objects
  Implemented as an array of instance variable count plus one elements. The first element is the object's class, while the remaining arguments are the instance variables.
- streams (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 75 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).

THE EVALUATOR

The process of evaluation in XLISP:

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

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

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

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

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

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

HOOK FUNCTIONS

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

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

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

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

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

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

LEXICAL CONVENTIONS

The following conventions must be followed when entering XLISP programs:

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

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

    ( ) ’ ‘ , " ;

and the escape characters:

    \ |

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

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

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

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

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

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

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

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

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

#\<char>	== the ASCII code of the printing character
#\newline	== ASCII linefeed character
#\space	== ASCII space character
#\rubout	== ASCII rubout (DEL)
#\C-<char>	== ASCII control character
#\M-<char>	== ASCII character with msb set (Meta character)
#\M-C-<char>	== ASCII control character with msb set

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

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

READTABLES

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

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

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

As an example, the following read macro allows the square brackets to be used as a more visibly appealing alternative to the SEND function:

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

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

XLISP defines several useful read macros:

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

SYMBOL CASE CONTROL

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

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

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

There are four major useful combinations of these modes:

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

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

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

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

C: *READTABLE-CASE* :PRESERVE

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

D: *READTABLE-CASE* :INVERT

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

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

LAMBDA LISTS

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

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

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

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

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

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

Here is the complete syntax for lambda lists:

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

where:

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

OBJECTS

Definitions:

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

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

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

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

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

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

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

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

THE 'Object' CLASS

Object THE TOP OF THE CLASS 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> default, or NIL, is *standard-output*, T is *terminal-io*
returns the object

:isnew — THE DEFAULT OBJECT INITIALIZATION ROUTINE
returns the object

:superclass — GET THE SUPERCLASS OF THE OBJECT
returns NIL
(Defined in classes.lsp, see :superclass below)

:ismemberof <class> — CLASS MEMBERSHIP
<class> class name
returns T if object member of class, else NIL
(defined in classes.lsp)

:iskindof <class> — CLASS MEMBERSHIP
<class> class name
returns T if object member of class or subclass of class, else NIL
(defined in classes.lsp)

:respondsto <sel> — SELECTOR KNOWLEDGE
<sel> message selector
returns T if object responds to message selector, else NIL.
(defined in classes.lsp)

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

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 16), otherwise initially unbound.
- object - constant, value is the class 'Object.'
- class - constant, value is the class 'Class'.
- internal-time-units-per-second - integer constant to divide returned times by to get time in seconds.
- pi - floating point aproximation of pi (constant defined when math extension is compiled).
- *obarray* - the object hash table. Length of array is a compilation option. Objects are hashed using the hash function and are placed on a list in the appropriate array slot.
- *terminal-io* - stream bound to keyboard and display. Do not alter.
- *standard-input* - the standard input stream, initially stdin. 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 3)
- *tracelist* - list of names of functions to trace, as set by trace function.
- *tracenable* - enable trace back printout on errors (see page 3).
- *tracelimit* - number of levels of trace back information (see page 3).
- *evalhook* - user substitute for the evaluator function (see page 7, and evalhook and applyhook functions).
- *applyhook* - user substitute for function application (see page 7, and evalhook and applyhook functions).
- *readtable* - the current readtable (see page 10).
- *unbound* - indicator for unbound symbols. A constant. Do not use this symbol since accessing any variable to which this has been bound will cause an unbound symbol error message.
- *gc-flag* - controls the printing of gc messages. When non-NIL, a message is printed after each garbage collection giving the total number of nodes and the number of nodes free.
- *gc-hook* - function to call after garbage collection (see page 7).
- *integer-format* - format for printing integers (when not bound to a string, defaults to "%d" or "%ld" depending on implementation)
- *ratio-format* - format for printing ratios (when not bound to a string, defaults to "%d/%d" or "%ld/%ld" depending on implementation)
- *float-format* - format for printing floats (when not bound to a string, defaults to "%g")

- *readtable-case* - symbol read and output case. See page 12 for details
- *print-case* - symbol output case when printing. See page 12 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 2).
- *displace-macros* - When not NIL, macros are replaced by their expansions when exectuted (see page 6).
- *random-state* - the default random-state used by the random function.

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

EVALUATION FUNCTIONS

(eval <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

(identity <expr>) RETURN THE EXPRESSION

New function. In common.lsp

<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

(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

SYMBOL FUNCTIONS

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

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

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

fsubr

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

(psetq [<sym> <expr>]...) PARALLEL VERSION OF SETQ

fsubr. All expressions are evaluated before any assignments are made.

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

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

fsubr

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

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

<value>	the new value
returns	the new value

(defsetf <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* property directly, although second format is not as efficient.

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

(push <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>)

(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>) MAKE AN INTERNED SYMBOL

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

(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

<sym>	the symbol
returns	the symbol's value

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

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

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

<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 integer)
returns	the hash index (integer 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

Defined in init.lsp

<sym>	the symbol
returns	the symbol

(unintern <sym>) UNINTERN A SYMBOL

Defined in common.lsp

<sym>	the symbol
returns	t if successful, NIL if symbol not interned

(defconstant <sym> <val>) DEFINE A CONSTANT

fsubr.

<sym>	the symbol
<val>	the value
returns	the value

(defparameter <sym> <val>) DEFINE A PARAMETER

fsubr.

<sym>	the symbol
<val>	the value
returns	the value

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

fsubr. Variable only initialized if not previously defined.

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

PROPERTY LIST FUNCTIONS

Note that property names are not limited to symbols.

(get <sym> <prop>) GET THE VALUE OF A PROPERTY

<sym> the symbol
<prop> the property symbol
returns the property value or NIL

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

<sym> the symbol
<val> the property value
<prop> the property symbol
returns the property value

(remprop <sym> <prop>) DELETE A PROPERTY

<sym> the symbol
<prop> the property symbol
returns NIL

HASH TABLE FUNCTIONS

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

(make-hash-table &key :size :test) MAKE A HASH TABLE

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

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

See also gethash in SETF.

<key>	hash key
<table>	hash table
<def>	value to return on no match (default is NIL)
returns	associated data, if found, or <def> if not found.

(remhash <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

See setf for setting elements of arrays

<array>	the array (or string)
<n>	the array index (integer, zero based)
returns	the value of the array element

(make-array <size>) MAKE A NEW ARRAY

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

<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

<type>	result type, one of CONS, LIST, ARRAY, STRING, or NIL
<fcn>	the function or function name
<expr>	a sequence for each argument of the function
returns	a new sequence of type <type>.

(every <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

<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>

(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) REMOVE ELEMENTS FROM A SEQUENCE

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

(remove-if <test> <seq> &key :key :start :end) REMOVE ELEMENTS THAT PASS TEST

(remove-if-not <test> <seq> &key :key :start :end) REMOVE ELEMENTS THAT FAIL TEST

<test>	the test predicate
<seq>	the sequence
:key	function to apply to test function argument (defaults to identity)
:start	starting index
:end	index of end+1, or NIL for (length <seq>)
returns	copy of sequence with matching or non-matching elements removed

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

COUNT ELEMENTS THAT PASS TEST

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

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

FIND FIRST ELEMENT THAT PASSES TEST

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

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

FIND POSITION OF FIRST ELEMENT THAT PASSES TEST

<test>	the test predicate
<seq>	the list
:key	function to apply to test function argument (defaults to identity)
:start	starting index
:end	index of end+1, or NIL for (length <seq>)
returns	position of first element of sequence that passes test, or NIL.

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

DELETE ELEMENTS FROM A SEQUENCE

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

(delete-if <test> <seq> &key :key :start :end)
DELETE ELEMENTS THAT PASS TEST

(delete-if-not <test> <seq> &key :key :start :end)
DELETE ELEMENTS THAT FAIL TEST

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

(reduce <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)
REMOVE DUPLICATES FROM SEQUENCE

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

(fill <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

LIST FUNCTIONS

(car <expr>) RETURN THE CAR OF A LIST NODE
<expr> the list node
returns the car of the list node

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

(first <expr>) A SYNONYM FOR CAR
(second <expr>) A SYNONYM FOR CADR
(third <expr>) A SYNONYM FOR CADDR
(fourth <expr>) A SYNONYM FOR CADDDR
(rest <expr>) A SYNONYM FOR CDR

(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

(last <list>) RETURN THE LAST LIST NODE OF A LIST
<list> the list
returns the last list node in the list

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

<list>	the list
<n>	count of elements to omit (default 1)
returns	copy of list with last element(s) absent.

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

<n>	the number of the element to return (zero origin)
<list>	the list
returns	the nth element or NIL if the list isn't that long

(nthcdr <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

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

(assoc <expr> <alist> &key :test :test-not :key)
FIND AN EXPRESSION IN AN A-LIST

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

(mapc <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CARS

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

(mapcar <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CARS

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

(mapl <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CDRS

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

(maplist <fcn> <list1> <list>...) APPLY FUNCTION TO SUCCESSIVE CDRS

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

(mapcan <fcn> <list1> <list>...) APPL FUNCTION TO SUCCESSIVE CARS

<fcn>	the function or function name
<listn>	a list for each argument of the function
returns	list of return values nconc'd together

(mapcon <fcn> <list1> <list>...) APPL FUNCTION TO SUCCESSIVE CDRS

<fcn>	the function or function name
<listn>	a list for each argument of the function
returns	list of return values nconc'd together

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

Does minimum copying as required by Common Lisp

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

(sublis <alist> <expr> &key :test :test-not :key) SUBSTITUTE WITH AN A-LIST

Does minimum copying as required by Common Lisp

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

(pairlis <keys> <values> [<alist>]) BUILD AN A-LIST FROM TWO LISTS

In file common.lsp

<keys>	list of association keys
<values>	list of association values, same length as keys
<alist>	existing association list, default NIL
returns	new association list

(copy-list <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>.

DESTRUCTIVE LIST FUNCTIONS

See also nreverse, delete, delete-if, delete-if-not, fill, and replace under SEQUENCE FUNCTIONS, setf under SYMBOL FUNCTIONS, and nintersection, nunion, nset-difference, and nset-exclusive-or under LIST FUNCTIONS.

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

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

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

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

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

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

(sort <list> <test> &key :key) SORT A LIST

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

ARITHMETIC FUNCTIONS

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

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

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

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

(truncate <expr> <denom>) TRUNCATES TOWARD ZERO
(round <expr> <denom>) ROUNDS TOWARD NEAREST 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. Results too big to be represented as integers are returned as floating point numbers as part of the math extension. Integers are returned as is.

<expr> the real number
<denom> real number to divide <expr> by before converting
returns the integer result of converting the number

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

<expr> the real number
returns the number as a floating point number

(+ [<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 math extension, division of integer numbers results in a rational quotient, rather than integer. To perform integer division, use TRUNCATE. When an integer complex is divided by an integer, the quotient is floating point complex.

<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.

(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.

<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 a fixnum, then the result type is the same as the type of x, unless fixnum or ratio and it would overflow, then the result type is a flonum.

(exp <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

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

Part of math extension
<expr> rational number
returns numerator of number (number if integer)

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

Part of math 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

(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

(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

<expr1>	the first integer
<expr2>	the second integer
returns	T if the result of the and operation is non-zero, else NIL

(ash <expr1> <expr2>) ARITHMETIC SHIFT

Part of math extension

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

STRING FUNCTIONS

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

(string <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

(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)

(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
	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.

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

(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

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

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

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

<chr>	the character
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 character

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

<chr>	the character
returns	the lower case character

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

<n>	the digit weight (integer)
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

(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.

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 <slot-desc>...)

or

(defstruct (name <option>...) <slot-desc>...)

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

The recognized options are:

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

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

Each slot description takes the form:

<name>

or

(<name> <defexpr>)

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

DEFSTRUCT causes access functions to be created for each of the slots and also arranges that SETF will work with those access functions. The access function names are constructed by taking the structure name, appending a '-' and then appending the slot name. This can be overridden by using the :CONC-NAME option.

DEFSTRUCT also makes a creation function called MAKE-<structname>, a copy function called COPY-<structname> and a predicate function called <structname>-P. The creation function takes keyword arguments for each of the slots. Structures can be created using the #S(read macro, as well.

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

For instance:

```
(defstruct foo bar (gag 2))
```

creates the following functions:

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

OBJECT FUNCTIONS

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

Messages defined for Object and Class are listed starting on page 17.

(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.

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

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

(defmethod <class> <sym> <fargs> <expr> ...) DEFINE A NEW METHOD

defined in class.lsp as a macro

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

(definst <class> <sym> [<args>...]) DEFINE A NEW GLOBAL INSTANCE

defined in class.lsp as a macro

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

PREDICATE FUNCTIONS

(atom <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.

(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 math extension.

<expr>	the expression to check
returns	t if the value is rational (integer or ratio), 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

(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

(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": eq: identical pointers -- works with characters, symbols, and arbitrarily small integers eql: works with all numbers, if same type (see also = on page 45) equal: lists and strings equalp: case insensitive characters (and strings), numbers of differing types, arrays (which can be equalp to string containing same elements)

(typep <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 78) or one of:

ATOM	any atom
NULL	NIL
LIST	matches NIL or any cons cell
STREAM	any stream
NUMBER	any number type
RATIONAL	fixnum or ratio (math extension)
STRUCT	any structure (except hash-table)
FUNCTION	any function, as defined by functionp (page 57)

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	

(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 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
<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 xlisp, the names are dynamically scoped.
<name> the block or function name (quoted symbol). If name is NIL, use function RETURN.
<value> the value to return (defaults to NIL)
returns never returns

(tagbody <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 xlisp, tags are dynamically scoped.
<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> [<eof> [<rflag>]]]) READ AN EXPRESSION

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

(set-macro-character <ch> <fcn> [T]) MODIFY READ TABLE

defined in init.lsp

<ch>	character to define
<fcn>	function to bind to character (see page 10)
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, *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

defined in common.lsp. Uses *terminal-io* stream for interaction.

<fmt>	optional format string for question (see page 67)
<arg>	arguments, if any, for format string
returns	T for yes, NIL for no.

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:

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

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

For ~A and ~S the full form is:

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

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

~15,,2,'.@A

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

For ~D the full form is:

~mincol,padchar@D

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

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

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

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

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

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

For ~T, the full form is:

~count,tabwidth@T

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

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

FILE I/O FUNCTIONS

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

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

When the transcript is active (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>]) READ A CHARACTER FROM A STREAM
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)
returns the character or NIL at end of file

(peek-char [<flag> [<stream>]]) PEEK AT THE NEXT CHARACTER
<flag> flag for skipping white space (default is NIL)
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)
returns the character or NIL at end of file

(write-char <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>]) READ A LINE FROM A STREAM
<stream> the input stream (default, or NIL, is *standard-input*, T is *terminal-io*)
returns the string excluding the #\newline, or NIL at end of file

(open <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 78). Files opened with type FIXNUM are binary files instead of ascii, which means no crlf to/from lf conversion takes place, and control-Z will not terminate an input file. It is the intent of Common Lisp that binary files only be accessed with read-byte and write-byte while ascii files be accessed with any function but read-byte and write-byte. XLISP does not enforce that distinction.
:if-exists	action to take if file exists. Argument ignored for :input (file is positioned at start) or :probe (file is closed)
:error	give error message
:rename	rename file to generated backup name, then open a new file of the original name. This is the default action
:new-version	same as :rename
:overwrite	file is positioned to start, original data intact
:append	file is positioned to end
:supersede	delete original file and open new file of the same name
:rename-and-delete	same as :supersede
NIL	close file and return NIL
:if-does-not-exist	action to take if file does not exist.
:error	give error message (default for :input, or :overwrite or :append)
:create	create a new file (default for :output or :io when not :overwrite or :append)
NIL	return NIL (default for :probe)
returns	a file stream, or sometimes NIL

(close <stream>) CLOSE A FILE STREAM

The stream becomes a "closed stream." Note that unbound file streams are closed automatically during a garbage collection.

<stream>	the stream, which may be a string stream
returns	t if stream closed, NIL if terminal (cannot be closed) or already closed.

(delete-file <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

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

<var> symbol name to bind stream to while evaluating expresssions (quoted)

<fname> file name string or symbol

<karg> keyword arguments for the implicit open command

<expr> expressions to evaluate while file is open (implicit progn)

returns value of last <expr>.

(read-byte [<stream>]) READ A BYTE FROM A STREAM

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

returns the byte (integer) or NIL at end of file

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

<byte> the byte to write (integer)

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

returns the byte (integer)

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

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

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

returns length of file, or NIL if cannot be determined.

(file-position <stream> [<expr>]) GET OR SET FILE POSITION

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

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

<expr> desired file position, if setting position. Can also be :start for start of file or :end for end of file.

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

STRING STREAM FUNCTIONS

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

<emsg>	the error message string
<arg>	the argument expression (printed after the message)
returns	never returns

(cerror <cmsg> <emsg> [<arg>]) SIGNAL A CORRECTABLE ERROR

<cmsg>	the continue message string
<emsg>	the error message string
<arg>	the argument expression (printed after the message)
returns	NIL when continued from the break loop

(break [<bmsg> [<arg>]]) ENTER A BREAK LOOP

<bmsg>	the break message string (defaults to "**BREAK**")
<arg>	the argument expression (printed after the message)
returns	NIL when continued from the break loop

(clean-up) CLEAN-UP AFTER AN ERROR

returns	never returns

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

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

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 number of segments to add, default 1
returns the number of segments added

(alloc <num> [<num2>]) CHANGE SEGMENT SIZE
<num> the number of nodes to allocate
<num2> the number of pointer elements to allocate in an array segment (when dynamic array allocation compiled). Default is no change.
returns the old number of nodes to allocate

(room) SHOW MEMORY ALLOCATION STATISTICS
Statistics (which are sent to *STANDARD-OUTPUT*) include:
Nodes - number of nodes, free and used
Free nodes - number of free nodes
Segments - number of node segments, including those reserved for characters and small integers.
Allocate - number of nodes to allocate in any new node segments
Total - total memory bytes allocated for node segments, arrays, and strings
Collections - number of garbage collections
When dynamic array allocation is compiled, the following additional statistics are printed:
Vector nodes - number of pointers in arrays and (size equivalent) strings
Vector segs - number of vector segments. Increases and decreases as needed.
Vec allocate - number of pointer elements to allocate in any new vector segment
returns NIL

(time <expr>) MEASURE EXECUTION TIME
fsubr.
<expr> the expression to evaluate
returns the result of the expression. The execution time is printed to *TRACE-OUTPUT*

(get-internal-real-time) GET 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 20). 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, fixnums can be coerced into characters or flonums. Ratios can be coerced into flonums. Flonums and ratios can be coerced into complex (so can fixnums, but they turn back into fixnums).

<expr>	the expression to coerce
<type>	desired type, as returned by type-of (see page 78)
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
	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 (integer)
returns	the value at the specified address (integer)

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

<addrs>	the address to poke (integer)
<value>	the value to poke into the address (integer)
returns	the value

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

<expr>	the node
returns	the address of the node (integer)

(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)

(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
structure	copy as an ARRAY

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

(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

(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	T, or NIL if fails

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.

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.

(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.

REPAIR.LSP

This file contains a structure editor.
Execute

(repair 'symbol) to edit a symbol.
(repairf symbol) to edit the function binding of a symbol (allows changing the argument list or function type, lambda or macro).

The editor alters the current selection by copying so that aborting all changes is generally posible; the exception is when editing a closure, if the closure is BACKed out of, the change is permanent.

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

Any array elements become lists when they are selected, and return to arrays upon RETURN or BACK commands.

Do not create new closures, because the environment will be incorrect. Closures become LAMBDA or MACRO expressions as the selection. Only the closure body may be changed; the argument list cannot be successfully modified, nor can the environment.

For class objects, only the methods and message names can be modified. For instance objects, instance variables can be examined (if the object under-stands the message :<ivar> for the particular ivar), and changed (if :SET-IVAR is defined for that class, as it is if CLASSES.LSP is used).

(command list on next page)

COMMANDS (general):

?	list available commands for the selection.
RETURN	exit, saving all changes.
ABORT	exit, without changes.
BACK	go back one level (as before CAR CDR or N commands).
B n	go back n levels.
L	display selection using pprint; if selection is symbol, give short description.
MAP	pprints each element of selection, or if selection is symbol then gives complete description of properties.
PLEV x	set *print-level* to x. (Initial default is *rep-print-level*)
PLEN x	set *print-length to x. (Initial default is *rep-print-length*)
EVAL x	evaluates x and prints result. The symbol @ is bound to the selection.
REPLACE x	replaces the current selection with evaluated x. The symbol @ is bound to the selection.

COMMANDS (if selection is symbol):

VALUE	edit the value binding.
FUNCTION	edit the function binding (must be a closure).
PROP x	edit property x.

COMMANDS (if selection is list):

CAR	select the CAR of the current selection.
CDR	select the CDR of the current selection.
n	where n is small non-negative integer, changes current selection to (NTH n list).
SUBST x y	all occurances of (quoted) y are replaced with (quoted) x. EQUAL is used for the comparison.
RAISE n	removes parenthesis surrounding nth element of selection.
LOWER n m	inserts parenthesis starting with the nth element, for m elements.
ARRAY n m	as in LOWER, but makes elements into an array.
I n x	inserts (quoted) x before nth element in selection.
R n x	replaces nth element in selection with (quoted) x.
D n	deletes nth element in selection.

All function names and global variables start with the string "rep-" or "*rep-*".

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.

User Interface Changes

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

-? 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

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.
ratio numbers -- new type.
complex numbers -- new type, can be integer or real.
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

New Variables and Constants

apply-hook Now activated
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).
print-level CL
print-length CL
random-state CL
ratio-format
readtable-case CL-like
terminal-io CL
internal-time-units-per-second CL
pi CL

New functions

ACONS CL*
ACOSH CL*
ADJOIN CL
APPLYHOOK 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)
COMPLEX CL
COMPLEXP CL
CONCATENATE CL
CONJUGATE CL
CONSTANTP CL
COPY-ALIST CL*
COPY-LIST CL*
COPY-TREE CL*
COSH CL*
COUNT-IF CL except no :from-end
DECF CL*
DEFCLASS * (define a new class)
DEFINST * (define a new instance)
DEFMETHOD * (define a new method)
DEFSETF CL*
DELETE-FILE CL
DENOMINATOR CL
DRAW (graphics -- MS/DOS only)
DRAWREL (graphics -- MS/DOS only)
ELT CL
EQUALP CL*
EVERY CL
FILE-LENGTH CL
FILE-POSITION CL
FILL CL*
FIND-IF CL except no :from-end
FLOOR CL
FRESH-LINE CL
FUNCTIONP CL*
GENERIC (implementation debugging function)

GET-INTERNAL-REAL-TIME CL
GET-INTERNAL-RUN-TIME CL
GETHASH CL
GOTO-XY (position cursor -- MS/DOS only)
HASH-TABLE-COUNT CL
IDENTITY CL*
IMAGPART CL
INCF CL*
INPUT-STREAM-P CL
INTERSECTION CL
LCM CL
LIST* CL
LOG CL
LOGTEST CL*
MAKE-HASK-TABLE CL
MAKE-RANDOM-STATE CL
MAP CL
MAPHASH CL
MODE (graphics -- MS/DOS only)
MOVE (graphics -- MS/DOS only)
MOVEREL (graphics -- MS/DOS only)
NINTERSECTION CL*
NOTANY CL
NOTEVERY CL
NREVERSE CL
NSET-DIFFERENCE CL*
NSET-EXCLUSIVE-OR CL*
NUMERATOR CL
NUNION CL*
OPEN-STREAM-P CL
OUTPUT-STREAM-P CL
PAIRLIS CL*
PHASE CL
POP CL*
POSITION-IF CL except no :from-end
PUSH CL*
PUSHNEW CL*
RATIONALP CL
REALPART CL
REDUCE CL except no :from-end
REMHASH CL
REMOVE-DUPLICATES CL except no :from-end
REPLACE CL*
ROUND CL
SEARCH CL except no :from-end
SET-DIFFERENCE CL
SET-EXCLUSIVE-OR CL*

SETF Placeform ELT CL
SETF Placeform GETHASH CL
SETF Placeform SEND* (set instance variable)
SIGNUM CL*
SINH CL*
SOME CL
SUBSETP CL
TANH CL*
TIME CL
TRUENAME CL
TYPEP CL
UNINTERN CL*
UNION CL
WITH-INPUT-FROM-STRING CL*
WITH-OPEN-FILE CL*
WITH-OUTPUT-TO-STRING CL*
Y-OR-N-P CL*

Changed functions

&ALLOW-OTHER-KEYS CL (now functions, is no longer ignored)
* 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)
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)
DEFVAR CL# (true special variables)
DELETE (added keywords :key :start :end. Works on arrays and strings)
DELETE-IF (added keywords :key :start :end. Works on arrays and strings)
DELETE-IF-NOT (added keywords :key :start :end. Works on arrays and strings)
EXP CL CR CX
EXPT CL CR CX

FORMAT (added directives ~D ~E ~F ~G ~& ~T ~\N and lowercase directives)
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-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)
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*)
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*)
READ-BYTE CL (input stream NIL is *standard-input*, T is *terminal-io*)
READ-CHAR CL (input stream NIL is *standard-input*, T is *terminal-io*)
READ-LINE CL (input stream NIL is *standard-input*, T is *terminal-io*)
REM CR CL (only two arguments now allowed, may be floating point)
REMOVE (added keywords :key :start :end. Works on arrays and strings)
REMOVE-IF (added keywords :key :start :end. Works on arrays and strings)
REMOVE-IF-NOT (added keywords :key :start :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)
SAVE (file name argument may be file stream)
SIN CL CR CX
SORT (added :key)
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)
WRITE-BYTE CL (output stream NIL is *standard-output*, T is *terminal-io*)
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 return NIL (or whatever value was supplied as the second argument to READ).

Once you are done reading from the file, you should close it. To close the file, use the following expression:

```
(close fp)
```

Evaluating this expression will cause the file to be closed.

Output to a File

Writing to a file is pretty much the same as reading from one. You need to open the file first. This time you should use the OPEN function to indicate that you will do output to the file. For example:

```
(setq fp (open "test.dat" :direction :output :if-exists :supersede))
```

Evaluating this expression will open the file "test.dat" for output. If the file already exists, its current contents will be discarded. If it doesn't already exist, it will be created. In any case, a FILE-STREAM object will be returned by the OPEN function. This file object will be assigned to the "fp" variable.

It is now possible to write to this file by supplying the value of the "fp" variable as the optional "stream" parameter in the PRINT function.

```
(print "Hello there" fp)
```

Evaluating this expression will result in the string "Hello there" being written to the file "test.dat". More data can be written to the file using the same technique.

Once you are done writing to the file, you should close it. Closing an output file is just like closing an input file.

```
(close fp)
```

Evaluating this expression will close the output file and make it permanent.

A Slightly More Complicated File Example

This example shows how to open a file, read each Lisp expression from the file and print it. It demonstrates the use of files and the use of the optional "stream" argument to the READ function.

```
(do* ((fp (open "test.dat" :direction :input))
      (ex (read fp) (read fp)))
     ((null ex) (close fp) nil)
  (print ex))
```

The file will be closed with the next garbage collection.

INDEX