# XLISP: An Experimental Object-oriented Language

Version 1.7

March 12, 1986

by
David Michael Betz
114 Davenport Ave.
Manchester, NH  03103

(603) 625-4691 (home)

# Table of Contents

# INTRODUCTION

XLISP is an experimental programming language combining some of the features of LISP with an object-oriented extension capability. It was implemented to allow experimentation with object-oriented programming on small computers. There are currently implementations running on the the VAX under VAX/VMS, on the 8088/8086 under MS-DOS, on the 68000 under CP/M-68K, on the Macintosh, on the Atari 520ST and on the Amiga. It is completely written in the programming language 'C' and is easily extended with user written built-in functions and classes. It is available in source form free of charge to non-commercial users.

Many traditional LISP functions are built into XLISP. 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. It assumes some knowledge of LISP and some understanding of the concepts of object-oriented programming.

A recommended text for learning LISP programming is the book "LISP" by Winston and Horn and published by Addison Wesley. The first edition of this book is based on MacLisp and the second edition is based on Common Lisp. Future versions of XLISP will continue to migrate towards compatibility with Common Lisp.

If you have any problems with XLISP, feel free to contact me for help or advice. Please remember that since XLISP is available in source form in a high level language, many users have been making versions available on a variety of machines. If you call to report a problem with a specific version, I may not be able to help you if that version runs on a machine to which I don't have access. Please have the version number of the version that you are running readily accessible before calling me.

If you find a bug in XLISP, first try to fix the bug yourself using the source code provided. If you are successful in fixing the bug, send the bug report along with the fix to me. If you don't have access to a C compiler or are unable to fix a bug, please send the bug report to me and I'll try to fix it.

Any suggestions for improvements will be welcomed. Feel free to extend the language in whatever way suits your needs. However, PLEASE DO NOT RELEASE ENHANCED VERSIONS WITHOUT CHECKING WITH ME FIRST!! I would like to be the clearing house for new features added to XLISP. If you want to add features for your own personal use, go ahead. But, if you want to distribute your enhanced version, contact me first. Please remember that the goal of XLISP is to provide a language to learn and experiment with LISP and object-oriented programming on small computers. I don't want it to get so big that it requires megabytes of memory to run.

## XLISP COMMAND LOOP

When XLISP is started, it first tries to load "init.lsp" from the default directory. It then loads any files named as parameters on the command line (after appending ".lsp" to their names). It then issues the following prompt:

```
>
```

This indicates that XLISP is waiting for an expression to be typed. When an incomplete expression has been typed (one where the left and right parens don't match) XLISP changes its prompt to:

```
n>
```

where n is an integer indicating how many levels of left parens remain unclosed.

When a complete expression has been entered, XLISP attempts to evaluate that expression. If the expression evaluates successfully, XLISP prints the result of the evaluation and then returns to the initial prompt waiting for another expression to be typed.

## 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 programmers.

- lists
- symbols
- strings
- integers
- floats
- objects
- arrays
- file pointers
- subrs (built-in functions)
- fsubrs (special forms)

Another data type is the stream. A stream is a list node whose car points to the head of a list of integers and whose cdr points to the last list node of the list. An empty stream is a list node whose car and cdr are nil. Each of the integers in the list represents a character in the stream. When a character is read from a stream, the first integer from the head of the list is removed and returned. When a character is written to a stream, the integer representing the character code of the character is appended to the end of the list. When a function indicates that it takes an input source as a parameter, this parameter can either be an input file pointer or a stream. Similarly, when a function indicates that it takes an output sink as a parameter, this parameter can either be an output file pointer or a stream.

# THE EVALUATOR

The process of evaluation in XLISP:

Integers, floats, strings, file pointers, subrs, fsubrs, objects and arrays evaluate to themselves

Symbols evaluate to the value associated with their current binding

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

If it is a subr, the remaining list elements are evaluated and the subr is called with these evaluated expressions as arguments.

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

If it is a list:

If the list is a function closure (a list whose car is a lambda expression and whose cdr is an environment list), the car of the list is used as the function to be applied and the cdr is used as the environment to be extended with the parameter bindings.

If the list is a lambda expression, the current environment is used for the function application.

In either of the above two cases, the remaining list elements are evaluated and the resulting expressions are bound to the formal arguments of the lambda expression. The body of the function is executed within this new binding environment.

If it is a list and the car of the list is 'macro', the remaining list elements are bound to the formal arguments of the macro expression. The body of the function is executed within this new binding environment. The result of this evaluation is considered the macro expansion. This result is then evaluated in place of the original expression.

If it is an object, the second list element is evaluated and used as a message selector. The message formed by

combining the selector with the values of the remaining list elements is sent to the object.

## 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.

Symbol names in XLISP can consist of any sequence of non-blank printable characters except the following:

```
( ) ' ` , " ;
```

Uppercase and lowercase characters are not distinguished within symbol names. All lowercase characters are mapped to uppercase on input.

Integer literals consist of a sequence of digits optionally beginning with a '+' 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.

Floating point literals consist of a sequence of digits optionally beginning with a '+' or '-' and including an embedded decimal point. The range of values a floating point number can represent is limited by the size of a C 'float' ('double' on machines with 32 bit addresses) on the machine on which XLISP is running.

Literal strings are sequences of characters surrounded by double quotes. 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
```

XLISP defines several useful read macros:

```
'<expr>          == (quote <expr>)
#'<expr>         == (function <expr>)
#(<expr>...)     == an array of the specified expressions
#x<hdigits>      == a hexadecimal number
```

```
#\<char> == the ASCII code of the character
`<expr>          == (backquote <expr>)
,<expr>          == (comma <expr>)
,@<expr>         == (comma-at <expr>)
```

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. It is an array with 128 entries, one for each of the ASCII character codes. Each entry contains one of the following things:

```
NIL              Indicating an invalid character
:CONSTITUENT     Indicating a symbol constituent
:WHITE-SPACE     Indicating a whitespace character
(:TMACRO . fun)  Terminating readmacro
(:NMACRO . fun)  Non-terminating readmacro
```

In the case of the last two forms, the "fun" component is a function definition. This can either be a pointer to a built-in readmacro function or a lambda expression. The function should take two parameters. The first is the input stream and the second is the character that caused the invocation of the readmacro. The character is passed as an integer. 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.

# 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. When the XLISP evaluator evaluates a list the value of whose first element is an object, it interprets the value of the second element of the list (which must be a symbol) as the message selector. The evaluator 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.

When a method is found, the evaluator binds the receiving object to the symbol 'self', binds the class in which the method was found to the symbol 'msgclass', 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.

THE 'Object' CLASS

Classes:

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

        :isnew  THE DEFAULT OBJECT INITIALIZATION ROUTINE
            returns     the object

        :sendsuper <sel> [<args>]...  SEND SUPERCLASS A MESSAGE
            <sel>       the message selector
            <args>      the message arguments
            returns     the result of sending the message

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 symbols
    <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
                  this list is of the form:
                    ([<farg>]...
                     [&optional [<oarg>]...]
                     [&rest <rarg>]
                     [&aux [<aux>]...])
                  where
                    <farg>   a formal argument
                    <oarg>   an optional argument
                    <rarg>   bound to rest of the arguments
                    <aux>    a auxiliary variable
    <code>      a list of executable expressions
    returns     the object
```

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.

When a new class is created by sending the ':new' message to the object 'Class', 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, class variables, and methods from its super-class.

SYMBOLS

- self - the current object (within a message context)
- msgclass - the class in which the current method was found
- *obarray* - the object hash table
- *standard-input* - the standard input file
- *standard-output* - the standard output file
- *breakenable* - flag controlling entering break loop on errors
- *tracenable* - enable baktrace on errors
- *tracelimit* - number of levels of trace back information
- *evalhook* - user substitute for the evaluator function
- *applyhook* - (not yet implemented)
- *readtable* - the current readtable
- *unbound* - indicator for unbound symbols
- *gc-flag* - controls the printing of gc messages

EVALUATION FUNCTIONS

(eval <expr>) EVALUATE AN XLISP EXPRESSION
    <expr> the expression to be evaluated
    returns the result of evaluating the expression

(apply <fun> <args>) APPLY A FUNCTION TO A LIST OF ARGUMENTS
    <fun> the function to apply (or function symbol)
    <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)
    <arg> arguments to pass to the function
    returns the result of calling the function with the arguments

(quote <expr>) RETURN AN EXPRESSION UNEVALUATED
    <expr> the expression to be quoted (quoted)
    returns <expr> unevaluated

(function <expr>) QUOTE A FUNCTION
    <expr> the function to be quoted (quoted)
    returns a function closure

(backquote <expr>) FILL IN A TEMPLATE
    <expr> the template
    returns a copy of the template with comma and comma-at
        expressions expanded

(lambda <args> [<expr>]...) MAKE A FUNCTION CLOSURE
    <args> the argument list (quoted)
    <expr> expressions of the function body
    returns the function closure

SYMBOL FUNCTIONS

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

(setf [<place> <expr>]...)  SET THE VALUE OF A FIELD
    <place>     the field specifier (quoted):
                    <sym>                set value of a symbol
                    (car <expr>)         set car of a list node
                    (cdr <expr>)         set cdr of a list node
                    (nth <n> <expr>)     set nth car of a list
                    (aref <expr> <n>)    set nth element of an array
                    (get <sym> <prop>)   set value of a property
                    (symbol-value <sym>) set value of a symbol
                    (symbol-plist <sym>) set property list of a
symbol
    <value>     the new value
    returns     the new value

(defun <sym> <fargs> [<expr>]...)  DEFINE A FUNCTION
(defmacro <sym> <fargs> [<expr>]...)  DEFINE A MACRO
    <sym>       symbol being defined (quoted)
    <fargs>     list of formal arguments (quoted)
                  this list is of the form:
                    ([<farg>]...
                     [&optional [<oarg>]...]
                     [&rest <rarg>]
                     [&aux [<aux>]...])
                  where
                    <farg>      is a formal argument
                    <oarg>      is an optional argument
                    <rarg>      bound to the rest of the arguments
                    <aux>       is an auxiliary variable
    <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
```

```
(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-plist <sym>)  GET THE PROPERTY LIST OF A SYMBOL
    <sym>       the symbol
    returns     the symbol's property list

(hash <sym> <n>)  COMPUTE THE HASH INDEX FOR A SYMBOL
    <sym>       the symbol or string
    <n>         the table size (integer)
    returns     the hash index (integer)
```

## PROPERTY LIST FUNCTIONS

(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>) REMOVE A PROPERTY
    <sym> the symbol
    <prop> the property symbol
    returns nil

## ARRAY FUNCTIONS

```
(aref <array> <n>)  GET THE NTH ELEMENT OF AN ARRAY
    <array>     the array
    <n>         the array index (integer)
    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
```

## 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

(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

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

(reverse <expr>) REVERSE A LIST
<expr> the list to reverse
returns a new list in the reverse order

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

(member <expr> <list> [<key> <test>]) FIND AN EXPRESSION IN A LIST
<expr> the expression to find
<list> the list to search
<key> the keyword :test or :test-not
<test> the test function (defaults to eql)
returns the remainder of the list starting with the expression

(assoc <expr> <alist> [<key> <test>]) FIND AN EXPRESSION IN AN A-LIST
<expr> the expression to find
<alist> the association list

```
<key>       the keyword :test or :test-not
<test>      the test function (defaults to eql)
returns     the alist entry or nil
```

(remove <expr> <list> [<key> <test>]) REMOVE AN EXPRESSION
<expr> the expression to delete
<list> the list
<key> the keyword :test or :test-not
<test> the test function (defaults to eql)
returns the list with the matching expressions deleted

(length <expr>) FIND THE LENGTH OF A LIST OR STRING
<expr> the list or string
returns the length of the list or string

(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

(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

```
(subst <to> <from> <expr> [<key> <test>])  SUBSTITUTE EXPRESSIONS
    <to>        the new expression
    <from>      the old expression
    <expr>      the expression in which to do the substitutions
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the expression with substitutions

(sublis <alist> <expr> [<key> <test>])  SUBSTITUTE WITH AN A-LIST
    <alist>     the association list
    <expr>      the expression in which to do the substitutions
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the expression with substitutions
```

DESTRUCTIVE 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

(delete <expr> <list> [<key> <test>]) DELETE AN EXPRESSION FROM A LIST
    <expr> the expression to delete
    <list> the list
    <key> the keyword :test or :test-not
    <test> the test function (defaults to eql)
    returns the list with the matching expressions deleted

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 expression is nil, nil otherwise

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

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

(boundp <sym>) IS THIS A BOUND SYMBOL?
    <sym> the symbol
    returns t if a value is bound to the symbol, nil otherwise

```
(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 NUMBER EVEN?
    <expr>      the number to test
    returns     t if the number is even, nil otherwise

(oddp <expr>)  IS THIS NUMBER ODD?
    <expr>      the number to test
    returns     t if the number is odd, nil otherwise

(eq <expr1> <expr2>)  ARE THE EXPRESSIONS IDENTICAL?
    <expr1>     the first expression
    <expr2>     the second expression
    returns     t if they are equal, nil otherwise

(eql <expr1> <expr2>)  ARE THE EXPRESSIONS IDENTICAL?
                         (WORKS WITH NUMBERS AND STRINGS)
    <expr1>     the first expression
    <expr2>     the second expression
    returns     t if they are equal, nil otherwise

(equal <expr1> <expr2>)  ARE THE EXPRESSIONS EQUAL?
    <expr1>     the first expression
    <expr2>     the second expression
    returns     t if they are equal, nil otherwise
```

CONTROL CONSTRUCTS

```
(cond [<pair>]...)  EVALUATE CONDITIONALLY
    <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
    <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
    <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>])  EXECUTE EXPRESSIONS CONDITIONALLY
    <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

(case <expr> [<case>]...)  SELECT BY CASE
    <expr>      the selection expression
    <case>      pair consisting of:
                    (<value> [<expr>]...)
                  where:
                    <value>     is a single expression or a list of
                                expressions (unevaluated)
                    <expr>      are expressions to execute if the
                                case matches
    returns     the value of the last expression of the matching case

(let ([<binding>]...) [<expr>]...)  CREATE LOCAL BINDINGS
(let* ([<binding>]...) [<expr>]...)  LET WITH SEQUENTIAL BINDING
    <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

(catch <sym> [<expr>]...)  EVALUATE EXPRESSIONS AND CATCH THROWS
    <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
    <sym>       the catch tag
    <expr>      the value for the catch to return (defaults to nil)
    returns     never returns
```

LOOPING CONSTRUCTS

```
(do ([<binding>]...) (<texpr> [<rexpr>]...) [<expr>]...)
(do* ([<binding>]...) (<texpr> [<rexpr>]...) [<expr>]...)
    <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> is the initial value of the symbol
                        <step> is 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
    <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)

(dotimes (<sym> <expr> [<rexpr>]) [<expr>]...)  LOOP FROM ZERO TO N-1
    <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)
```

# THE PROGRAM FEATURE

```
(prog ([<binding>]...) [<expr>]...)  THE PROGRAM FEATURE
(prog* ([<binding>]...) [<expr>]...)  PROG WITH SEQUENTIAL BINDING
    <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

(go <sym>)  GO TO A TAG WITHIN A PROG CONSTRUCT
    <sym>       the tag (quoted)
    returns     never returns

(return [<expr>])  CAUSE A PROG CONSTRUCT TO RETURN A VALUE
    <expr>      the value (defaults to nil)
    returns     never returns

(prog1 <expr1> [<expr>]...)  EXECUTE EXPRESSIONS SEQUENTIALLY
    <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
    <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
    <expr>      the expressions to evaluate
    returns     the value of the last expression (or nil)
```

DEBUGGING AND ERROR HANDLING

```
(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

(continue)  CONTINUE FROM A CORRECTABLE ERROR
    returns     never returns

(errset <expr> [<pflag>])  TRAP ERRORS
    <expr>      the expression to execute
    <pflag>     flag to control printing of the error message
    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>     the value for *evalhook*
    <ahook>     the value for *applyhook*
    <env>       the environment (default is nil)
    returns     the result of evaluating the expression
```

ARITHMETIC FUNCTIONS

(truncate <expr>) TRUNCATES A FLOATING POINT NUMBER TO AN INTEGER
    <expr> the number
    returns the result of truncating the number

(float <expr>) CONVERTS AN INTEGER TO A FLOATING POINT NUMBER
    <expr> the number
    returns the result of floating the integer

(+ <expr>...) ADD A LIST OF NUMBERS
    <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
    <expr> the numbers
    returns the result of the multiplication

(/ <expr>...) DIVIDE A LIST OF NUMBERS
    <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
    <expr> the numbers
    returns the result of the remainder operation

(min <expr>...) THE SMALLEST OF A LIST OF NUMBERS
    <expr> the expressions to be checked
    returns the smallest number in the list

(max <expr>...) THE LARGEST OF A LIST OF NUMBERS
    <expr> the expressions to be checked
    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

(random <n>)  COMPUTE A RANDOM NUMBER BETWEEN 1 and N-1
    <n>         the upper bound (integer)
    returns     a random number
```

```
(sin <expr>)  COMPUTE THE SINE OF A NUMBER
    <expr>      the floating point number
    returns     the sine of the number

(cos <expr>)  COMPUTE THE COSINE OF A NUMBER
    <expr>      the floating point number
    returns     the cosine of the number

(tan <expr>)  COMPUTE THE TANGENT OF A NUMBER
    <expr>      the floating point number
    returns     the tangent of the number

(expt <x-expr> <y-expr>)  COMPUTE X TO THE Y POWER
    <x-expr>    the floating point number
    <y-expr>    the floating point exponent
    returns     x to the y power

(exp <x-expr>)  COMPUTE E TO THE X POWER
    <x-expr>    the floating point number
    returns     e to the x power

(sqrt <expr>)  COMPUTE THE SQUARE ROOT OF A NUMBER
    <expr>      the floating point number
    returns     the square root of the number
```

BITWISE LOGICAL FUNCTIONS

(log-and <expr>...) THE BITWISE AND OF A LIST OF NUMBERS
    <expr> the numbers
    returns the result of the and operation

(log-ior <expr>...) THE BITWISE INCLUSIVE OR OF A LIST OF NUMBERS
    <expr> the numbers
    returns the result of the inclusive or operation

(log-xor <expr>...) THE BITWISE EXCLUSIVE OR OF A LIST OF NUMBERS
    <expr> the numbers
    returns the result of the exclusive or operation

(log-not <expr>) THE BITWISE NOT OF A NUMBER
    <expr> the number
    returns the bitwise inversion of number

## RELATIONAL FUNCTIONS

The relational functions can be used to compare integers, floating point numbers or strings.

(< <e1> <e2>) TEST FOR LESS THAN
    <e1> the left operand of the comparison
    <e2> the right operand of the comparison
    returns the result of comparing <e1> with <e2>

(<= <e1> <e2>) TEST FOR LESS THAN OR EQUAL TO
    <e1> the left operand of the comparison
    <e2> the right operand of the comparison
    returns the result of comparing <e1> with <e2>

(= <e1> <e2>) TEST FOR EQUAL TO
    <e1> the left operand of the comparison
    <e2> the right operand of the comparison
    returns the result of comparing <e1> with <e2>

(/= <e1> <e2>) TEST FOR NOT EQUAL TO
    <e1> the left operand of the comparison
    <e2> the right operand of the comparison
    returns the result of comparing <e1> with <e2>

(>= <e1> <e2>) TEST FOR GREATER THAN OR EQUAL TO
    <e1> the left operand of the comparison
    <e2> the right operand of the comparison
    returns the result of comparing <e1> with <e2>

(> <e1> <e2>) TEST FOR GREATER THAN
    <e1> the left operand of the comparison
    <e2> the right operand of the comparison
    returns the result of comparing <e1> with <e2>

STRING 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

(string <expr>)  MAKE A STRING FROM AN INTEGER ASCII VALUE
    <expr>      the integer
    returns     a one character string

(strcat [<expr>]...)  CONCATENATE STRINGS
    <expr>      the strings to concatenate
    returns     the result of concatenating the strings

(substr <expr> <sexpr> [<lexpr>]) EXTRACT A SUBSTRING
    <expr>      the string
    <sexpr>     the starting position
    <lexpr>     the length (default is rest of string)
    returns     substring starting at <sexpr> for <lexpr>
```

## INPUT/OUTPUT FUNCTIONS

```
(read [<source> [<eof> [<rflag>]]])  READ AN XLISP EXPRESSION
    <source>    the input source (default is standard input)
    <eof>       the value to return on end of file (default is nil)
    <rflag>     recursive read flag (default is nil)
    returns     the expression read

(print <expr> [<sink>])  PRINT A LIST OF VALUES ON A NEW LINE
    <expr>      the expressions to be printed
    <sink>      the output sink (default is standard output)
    returns     the expression

(prin1 <expr> [<sink>])  PRINT A LIST OF VALUES
    <expr>      the expressions to be printed
    <sink>      the output sink (default is standard output)
    returns     the expression

(princ <expr> [<sink>])  PRINT A LIST OF VALUES WITHOUT QUOTING
    <expr>      the expressions to be printed
    <sink>      the output sink (default is standard output)
    returns     the expression

(terpri [<sink>])  TERMINATE THE CURRENT PRINT LINE
    <sink>      the output sink (default is standard output)
    returns     nil

(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
```

FILE I/O FUNCTIONS

```
(openi <fname>)  OPEN AN INPUT FILE
    <fname>     the file name string or symbol
    returns     a file pointer

(openo <fname>)  OPEN AN OUTPUT FILE
    <fname>     the file name string or symbol
    returns     a file pointer

(close <fp>)  CLOSE A FILE
    <fp>        the file pointer
    returns     nil

(read-char [<source>])  READ A CHARACTER FROM A FILE OR STREAM
    <source>    the input source (default is standard input)
    returns     the character (integer)

(peek-char [<flag> [<source>]])  PEEK AT THE NEXT CHARACTER
    <flag>      flag for skipping white space (default is nil)
    <source>    the input source (default is standard input)
    returns     the character (integer)

(write-char <ch> [<sink>])  WRITE A CHARACTER TO A FILE OR STREAM
    <ch>        the character to put (integer)
    <sink>      the output sink (default is standard output)
    returns     the character (integer)

(read-line [<source>])  READ A LINE FROM A FILE OR STREAM
    <source>    the input source (default is standard input)
    returns     the input string
```

SYSTEM FUNCTIONS

```
(load <fname> [<vflag> [<pflag>]])  LOAD AN XLISP SOURCE FILE
    <fname>     the filename string or symbol
    <vflag>     the verbose flag (default is t)
    <pflag>     the print flag (default is nil)
    returns     the filename

(transcript [<fname>])  CREATE A FILE WITH A TRANSCRIPT OF A SESSION
    <fname>     file name string or symbol
                (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
    returns     the number of segments added

(alloc <num>)  CHANGE NUMBER OF NODES TO ALLOCATE IN EACH SEGMENT
    <num>       the number of nodes to allocate
    returns     the old number of nodes to allocate

(mem)  SHOW MEMORY ALLOCATION STATISTICS
    returns     nil

(type-of <expr>)  RETURNS THE TYPE OF THE EXPRESSION
    <expr>      the expression to return the type of
    returns     nil if the value is nil otherwise one of the symbols:
                  :SYMBOL for symbols
                  :OBJECT for objects
                  :CONS   for conses
                  :SUBR   for built-ins with evaluated arguments
                  :FSUBR  for built-ins with unevaluated arguments
                  :STRING for strings
                  :FIXNUM for integers
                  :FLONUM for floating point numbers
                  :FILE   for file pointers
                  :ARRAY  for arrays

(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)
```

(exit) EXIT XLISP
    returns never returns