XLISP: An Object-Oriented Lisp

David Michael Betz

October 25, 2013

Contents

1 xlisp

XLISP: An Object-Oriented Lisp

Version 3.0 January 28, 1997

David Michael Betz 18 Garrison Drive Bedford, NH 03110

(603) 472-2389 (home)

1.1 Introduction

XLISP is a dialect of the Lisp programming language with extensions to support object-oriented programming. Over the years I have released many versions of XLISP with differing goals. The original XLISP was written to run on tiny machines like the Z80 under CP/M and hence was a small language. When Common Lisp came around, I made an attempt to bring XLISP into conformance with that standard. Later I learned about Scheme,

an elegant dialect of Lisp designed by Gerald Sussman and Guy Steele Jr. and realized that it more closely matched my desire for a small but powerful language. This version of XLISP is mostly based on Scheme with my own object system and some Common Lispish extensions. It is derived from XScheme, my earlier implementation of Scheme.

Unlike the original XLISP, this version compiles to bytecodes instead of directly interpreting the source s-expressions. This makes it somewhat faster than the old XLISP for normal functions and much faster for macros since they are expanded at compile time not run time.

There are currently implementations of XLISP running on the IBM-PC and clones under MS-DOS, on the Macintosh and under OS/2. It is completely written in ANSI C and is easily extended with user written built-in functions and classes. It is available in source form to non-commercial users.

This document is a brief description of XLISP. XLISP follows the "Revised^3 Report on the Algorithmic Language Scheme" with extensions. It assumes some knowledge of Scheme or LISP and some understanding of the concepts of object-oriented programming.

I recommend the book "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Jay Sussman and published by MIT Press and the McGraw-Hill Book Company for learning Scheme (and programming in general). You might also find "The Scheme Programming Language" by R. Kent Dybvig and "The Little Lisper" by Daniel P. Friedman and Matthias Felleisen to be helpful.

1.2 A Note from the Author

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 I don't have access to. Please have the version number of XLISP 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.

1.3 Constants

#T #!TRUE

The true value. Where boolean expressions are required, any value other than #F is interpreted as a true value.

#F #!FALSE

The false value. In XLISP, false and the empty list are the same value.

1.4 Built-In Variables

PACKAGE

Bound to the default package.

READTABLE

Bound to the current read table.

ERROR-HANDLER

Bound to a function to handle errors. The function should take two arguments, the function where the error occured and the environment at the time of the error. It shouldn't return.

UNBOUND-HANDLER

Bound to a function to handle unbound symbol errors. The function should take two arguments, the symbol that is unbound and a continuation to call after correcting the error.

LOAD-PATH

Bound to the path used by the LOAD function. This is initialized to the contents of the XLISP environment variable or, if that is not defined, to the path where the XLISP executable was found. The value consists of a list of strings that should be paths to directories XLISP should search for files being loaded. Each string should end with an appropriate directory terminator (the backslash under MS-DOS, the slash under UNIX or a colon on the Macintosh.

STANDARD-INPUT

Bound to the standard input port.

STANDARD-OUTPUT

Bound to the standard output port.

ERROR-OUTPUT

Bound to the error output port.

FIXNUM-FORMAT

A printf style format string for printing fixed point numbers. FIXNUMs are generally represented by long integers so this should usually be set to "%ld".

HEXNUM-FORMAT

A printf style format string for printing fixed point numbers in hexadecimal. FIXNUMs are generally represented by long integers so this should usually be set to "%lx".

FLONUM-FORMAT

A printf style format string for printing floating point numbers. This is usually set to "%.15g".

PRINT-CASE

Bound to a symbol that controls the case in which symbols are printed. Can be set to UPCASE or DOWNCASE.

SOFTWARE-TYPE

Bound to a symbol that indicates the host software. The following types are defined currently:

win95 Windows 95 dos32 Command line DOS under Windows 95 unix Unix or Linux mac Macintosh T

Bound to #t.

NIL

Bound to the empty list.

OBJECT

Bound to the class "Object".

CLASS

Bound to the class "Class".

1.5 Expressions

variable

An expression consisting of a variable is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

(QUOTE datum) 'datum

(QUOTE datum) evaluates to datum. Datum may be any external representation of an XLISP value. This notation is used to include literal constants in XLISP code. (QUOTE datum) may be abbreviated as 'datum. The two notations are equivalent in all respects.

constant

Numeric constants, string constants, character constants and boolean constants evaluate "to themselves"; they need not be quoted.

(operator operand...)

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated and the resulting procedure is passed the resulting arguments.

(object selector operand...)

A message sending form is written by enclosing in parentheses expressions for the receiving object, the message selector, and the arguments to be passed to the method. The receiver, selector, and argument expressions are evaluated, the message selector is used to select an appropriate method to handle the message, and the resulting method is passed the resulting arguments. (LAMBDA formals body)

Formals should be a formal argument list as described below, and body should be a sequence of one or more expressions. A lambda expression evaluates to a procedure. The environment in effect when the lambda expression is evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list to fresh locations, the corresponding actual argument values will be stored in those locations, and the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The result of the last expression in the body will be returned as the result of the procedure call.

Formals should have the following form:

(var... [#!OPTIONAL ovar...] [. rvar])

or

(var... [#!OPTIONAL ovar...] [#!REST rvar])

where:

var is a required argument ovar is an optional argument rvar is a "rest" argument

There are three parts to a formals list. The first lists the required arguments of the procedure. All calls to the procedure must supply values for each of the required arguments. The second part lists the optional arguments of the procedure. An optional argument may be supplied in a call or omitted. If it is omitted, a special value is given to the argument that satisfies the default-object? predicate. This provides a way to test to see if an optional argument was provided in a call or omitted. The last part of the formals list gives the "rest" argument. This argument will be bound to the

rest of the list of arguments supplied to a call after the required and optional arguments have been removed.

Alternatively, you can use Common Lisp syntax for the formal parameters:

(var... [&optional {ovar | (ovar [init [svar]])}... [&rest rvar] [&key {kvar | ({kvar | (key kvar)} [init [svar]])}... [&aux {avar | (avar [init])}])

where:

var is a required argument ovar is an optional argument rvar is a "rest" argument kvar is a keyword argument avar is an aux variable svar is a "supplied-p" variable

See "Common Lisp, the Language" by Guy Steele Jr. for a description of this syntax.

(NAMED-LAMBDA name formals body)

NAMED-LAMBDA is the same as LAMBDA except that the specified name is associated with the procedure.

(IF test consequent [alternate])

An if expression is evaluated as follows: first, test is evaluated. If it yields a true value, then consequent is evaluated and its value is returned. Otherwise, alternate is evaluated and its value is returned. If test yields a false value and no alternate is specified, then the result of the expression is unspecified.

A false value is nil or the empty list. Every other value is a true value.

(SET! variable expression)

Expression is evaluated, and the resulting value is stored in the location to which variable is bound. Variable must be bound in some region or at the top level. The result of the set! expression is unspecified.

(COND clause...)

Each clause should be of the form

(test expression...)

where test is any expression. The last clause may be an "else clause," which has the form

(ELSE expression...)

A cond expression is evaluated by evaluating the test expressions of successive clauses in order until one of them evaluates to a true value. When a test evaluates to a true value, then the remaining expressions in its clause are evaluated in order, and the result of the last expression in the clause is returned as the result of the entire cond expression. If the selected clause contains only the test and no expressions, then the value of the test is returned as the result. If all tests evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is

an else clause, then its expressions are evaluated, and the value of the last one is returned.

(AND test...)

The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

(OR test...)

The test expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.

1.6 Multiple Values

(VALUES expr...)

The results of evaluating this expression are the values of the expressions given as arguments. It is legal to pass no values as in (VALUES) to indicate no values.

(VALUES-LIST list)

The results of evaluating this expression are the values in the specified list.

(MULTIPLE-VALUE-BIND (var...) vexpr expr...)

The multiple values produced by vexpr are bound to the specified variables and the remaining expressions are evaluated in an environment that includes those variables.

(MULTIPLE-VALUE-CALL function expr)

The multiple values of expr are passed as arguments to the specified function.

1.7 Non-Local Exits

(CATCH tag expr...)

Evaluate the specified expressions in an environment where the specified tag is visible as a target for THROW. If no throw occurs, return the value(s) of the last expression. If a throw occurs that matches the tag, return the value(s) specified in the THROW form.

(THROW tag expr...)

Throw to a tag established by the CATCH form. In the process of unwinding the stack, evaluate any cleanup forms associated with UNWIND-PROTECT forms established between the target CATCH and the THROW form.

(THROW-ERROR arg)

Throw an error. This is basically equivilent to (THROW 'ERROR arg) except that care is taken to make sure that recursive errors are not produced if there is no corresponding CATCH for the 'ERROR tag.

(UNWIND-PROTECT pexpr expr...)

Evaluate pexpr (the protected expression) and then the other expressions and return the value(s) of pexpr. If an error or a THROW occurs during the evaluation of the protected form, the other expressions (known as cleanup forms) are evaluated during the unwind process.

1.8 Binding Forms

(LET [name] bindings body)

Bindings should have the form

((variable init)...)

where each init is an expression, and body should be a sequence of one or more expressions. The inits are evaluated in the current envirnoment, the variables are bound to fresh locations holding the results, the body is evaluated in the

If a name is supplied, a procedure that takes the bound variables as its arguments and has the body of the LET as its body is bound to that name.

(LET* bindings body)

Same as LET except that the bindings are done sequentially from left to right and the bindings to the left are visible while evaluating the initialization expressions for each variable.

(LETREC bindings body)

Bindings should have the form

((variable init)...)

and body should be a sequence of one or more expressions. The variables are bound to fresh locations holding undefined values; the inits are evaluated in the resulting environment; each variable is assigned to the result of the corresponding init; the body is evaluated in the resulting environment; and the value of the last expression in body is returned. Each binding of a variable has the entire letrec expression as its region, making it possible to define mutually recursive procedures. One restriction of letrec is very important: it must be possible to evaluate each init without referring to the value of

any variable. If this restriction is violated, then the effect is undefined, and an error may be signalled during evaluation of the inits. The restriction is necessary because XLISP passes arguments by value rather than by name. In the most common uses of letrec, all the inits are lambda expressions and the restriction is satisfied automatically.

1.9 Sequencing

(BEGIN expression...) (SEQUENCE expression...)

The expressions are evaluated sequentially from left to right, and the value of the last expression is returned. This expression type is used to sequence side effects such as input and output.

1.10 Delayed Evaluation

(CONS-STREAM expr1 expr2)

Create a cons stream whose head is expr1 (which is evaluated immediately) and whose tail is expr2 (whose evaluation is delayed until TAIL is called).

(HEAD expr)

Returns the head of a stream.

(TAIL expr)

Returns the tail of a stream by calling FORCE on the promise created by CONS-STREAM.

(DELAY expression)

Evaluating this expression creates a "promise" to evaluate expression at a later time.

(FORCE promise)

Applying FORCE to a promise generated by DELAY requests that the promise produce the value of the expression passed to DELAY. The first time a promise is FORCEed, the DELAY expression is evaluated and the value stored. On subsequent calls to FORCE with the same promise, the saved value is returned.

1.11 Iteration

(WHILE test expression...)

While is an iteration construct. Each iteration begins by evaluating test; if the result is false, then the loop terminates and the value of test is returned as the value of the while expression. If test evaluates to a true value, then the expressions are evaluated in order for effect and the next iteration begins.

1.12 Definitions

(DEFINE variable expression)

Define a variable and give it an initial value.

(DEFINE (variable . formals) body)

Define a procedure. Formals should be specified in the same way as with LAMBDA.

(DEFINE-MACRO (name . formals) body)

Defines a macro with the specified name.

1.13 The Object System

XLISP provides a fairly simple single inheritance object system. Each object is an instance of a class and classes themselves are objects. Each object has a set of instance variables where it stores its private data and in addition has access to a set of class variables that are shared amongst all instances of the same class.

Each class has a set of methods with which it responds to messages sent to its instances. A message is sent using a syntax similar to a function call:

(object selector expr...)

Where object is the object receiving the message, selector is a symbol used to select the appropriate method for handling the message and the expressions are arguments to pass to the method. A method may return zero or more values. Within a method, the object's instance variables and class variables are bound as if they were lexical variables.

(DEFINE-CLASS name decl...)

Creates a class with the specified class name and binds the global variable with that name to the new class.

Decl is:

(SUPER-CLASS super)

Specifies the single superclass. If not specified, the superclass is Object.

(INSTANCE-VARIABLES ivar...) (IVARS ivar...)

Specifies the instance variables of the new class.

(CLASS-VARIABLES {cvar | (cvar init)}...) (CVARS {cvar | (cvar init)}...)

Specifies the class variables of the new class.

(DEFINE-METHOD (class selector formals) expr...)

Defines a method for the specified class with the specified selector. Within a method, the symbol self refers to the object receiving the message. Also, all instance variables and class variables are available as if they

were lexical variables.

(DEFINE-CLASS-METHOD (class selector formals) expr...)

Defines a class method for the specified class with the specified selector. Within a method, the symbol self refers to the class receiving the message. Also class variables are available as if they were lexical variables.

(SUPER selector expr...)

When used within a method, sends a message to the superclass of the class where the current method was found.

1.14 Methods for the Built-In Classes

Class:

(Class 'make-instance)

Make an uninitialize instance of a class.

(Class 'new &rest args)

Make and initialize an instance of a class. The new instance is initialized by sending it the 'initialize message with the arguments passed to 'new. The result of the 'initialize method is returned as the result of 'new. The 'initialize method should return self as its value.

(Class 'initialize ivars &optional cvars super name)

Default class initialization method.

(Class 'answer selector formals body)

Add a method to a class.

(Class 'show &optional port)

Display information about a class.

Object:

(Object 'initialize)

Default initialization method.

(Object 'class)

Return the class of an object.

(Object 'get-variable var)

Get the value of an instance variable.

(Object 'set-variable! var expr)

Set the value of an instance variable.

(Object 'show &optional port)

Display information about an object.

1.15 List Functions

(CONS expr1 expr2)

Create a new pair whose car is expr1 and whose cdr is expr2.

(ACONS key data alist)

Is equivilent to (CONS (CONS key data) alist) and is used to add a pair to an association list.

(CAR pair) (FIRST pair)

Extract the car of a pair.

(CDR pair) (REST pair)

Extract the cdr of a pair.

(CxxR pair) (CxxxR pair) (CxxxxR pair)

These functions are short for combinations of CAR and CDR. Each 'x' is stands for either 'A' or 'D'. An 'A' stands for the CAR function and a 'D' stands for the CDR function. For instance, (CADR x) is the same as (CAR (CDR x)).

(SECOND list) (THIRD list) (FOURTH list)

Extract the specified elements of a list.

(LIST expr...)

Create a list whose elements are the arguments to the function. This function can take an arbitrary number of arguments. Passing no arguments results in the empty list.

(LIST* expr...)

Create a list whose elements are the arguments to the function except that the last argument is used as the tail of the list. This means that the call (LIST* 1 2 3) produce the result (1 2 . 3).

(APPEND list...)

Append lists to form a single list. This function takes an arbitrary number of arguments. Passing no arguments results in the empty list.

(REVERSE list)

Create a list whose elements are the same as the argument except in reverse order.

(LAST-PAIR list)

Return the last pair in a list.

(LENGTH list)

Compute the length of a list.

(PAIRLIS keys data &optional alist)

Creates pairs from corresponding elements of keys and data and pushes these onto alist. For instance:

(pairlis '(x y) '(1 2) '((z . 3))) => ((x . 1) (y . 2) (z . 3))

(COPY-LIST list)

Makes a top level copy of the list.

(COPY-TREE list)

Make a deep copy of a list.

(COPY-ALIST alist)

Copy an association list by copying each top level pair in the list.

(END? list)

Returns #f for a pair, #t for the empty list and signals an error for all other types.

(MEMBER expr list) (MEMV expr list) (MEMQ expr list)

Find an element in a list. Each of these functions searches the list looking for an element that matches expr. If a matching element is found, the remainder of the list starting with that element is returned. If no matching element is found, the empty list is returned. The functions differ in the test used to determine if an element matches expr. The MEMBER function uses EQUAL?, the MEMV function uses EQV? and the MEMQ function uses EQ?.

(ASSOC expr alist) (ASSV expr alist) (ASSQ expr alist)

Find an entry in an association list. An association list is a list of pairs. The car of each pair is the key and the cdr is the value. These functions search an association list for a pair whose key matches expr. If a matching pair is found, it is returned. Otherwise, the empty list is returned. The functions differ in the test used to determine if a key matches expr. The ASSOC function uses EQUAL?, the ASSV function uses EQV? and the ASSQ function uses EQ?.

(LIST-REF list n)

Return the nth element of a list (zero based).

(LIST-TAIL list n)

Return the sublist obtained by removing the first n elements of list.

1.16 Destructive List Functions

(SET-CAR! pair expr)

Set the car of a pair to expr. The value returned by this procedure is unspecified.

(SET-CDR! pair expr)

Set the cdr of a pair to expr. The value returned by this procedure is unspecified.

(APPEND! list...)

Append lists destructively.

1.17 Sequence Functions

At the moment these sequence functions work only with lists.

(MAPCAR)
(MAPC)
(MAPCAN)
(MAPLIST)
(MAPL)
(MAPCON)
(SOME)
(EVERY)
(NOTANY)
(NOTEVERY)
(FIND)
(FIND-IF)
(FIND-IF-NOT)
(MEMBER)
(MEMBER-IF)
(MEMBER-IF-NOT)
(ASSOC)
(ASSOC-IF)
(ASSOC-IF-NOT)
(RASSOC)
(RASSOC-IF)
(RASSOC-IF-NOT)
(REMOVE)
(REMOVE-IF)
(REMOVE-IF-NOT)
(DELETE)
(DELETE-IF)
(DELETE-IF-NOT)
(COUNT)
(COUNT-IF)
(COUNT-IF-NOT)
(POSITION)
(POSITION-IF)
(POSITION-IF-NOT)

1.18 Symbol Functions

(BOUND? sym [env])

Returns #t if a global value is bound to the symbol and #f otherwise.

(SYMBOL-NAME sym)

Get the print name of a symbol.

(SYMBOL-VALUE sym [env])

Get the global value of a symbol.

(SET-SYMBOL-VALUE! sym expr [env])

Set the global value of a symbol. The value returned by this procedure is unspecified.

(SYMBOL-PLIST sym)

Get the property list associated with a symbol.

(SET-SYMBOL-PLIST! sym plist)

Set the property list associate with a symbol. The value returned by this procedure is unspecified.

(SYMBOL-PACKAGE sym)

Returns the package containing the symbol.

(GENSYM &optional sym | str | num)

Generate a new, uninterned symbol. The print name of the symbol will consist of a prefix with a number appended. The initial prefix is "G" and the initial number is 1. If a symbol is specified as an argument, the prefix is set to the print name of that symbol. If a string is specified, the prefix is set to that string. If a number is specified, the numeric suffix is set to that number. After the symbol is generated, the number is incremented so subsequent calls to GENSYM will generate numbers in sequence.

(GET sym prop)

Get the value of a property of a symbol. The prop argument is a symbol that is the property name. If a property with that name exists on the symbols property list, the value of the property is returned. Otherwise, the empty list is returned.

(PUT sym prop expr)

Set the value of a property of a symbol. The prop argument is a symbol that is the property name. The property/value combination is added to the property list of the symbol.

(REMPROP sym prop)

Remove the specified property from the property list of the symbol.

1.19 Package Functions

(MAKE-PACKAGE name &key uses)
(FIND-PACKAGE name)
(LIST-ALL-PACKAGES)
(PACKAGE-NAME pack)
(PACKAGE-NICKNAMES pack)
(IN-PACKAGE pack)
(USE-PACKAGE name [pack])
(UNUSE-PACKAGE name [pack])
(PACKAGE-USE-LIST pack)
(PACKAGE-USED-BY-LIST pack)
(EXPORT sym [pack])
(UNEXPORT sym [pack])
(IMPORT sym [pack])
(INTERN pname [pack])
(UNINTERN sym [pack])
(MAKE-SYMBOL pname)
(FIND-SYMBOL sym [pack])

1.20 Vector Functions

(VECTOR expr...)

Create a vector whose elements are the arguments to the function. This function can take an arbitrary number of arguments. Passing no arguments results in a zero length vector.

(MAKE-VECTOR len)
Make a vector of the specified length.
(VECTOR-LENGTH vect)
Get the length of a vector.
(VECTOR-REF vect n)
Return the nth element of a vector (zero based).
(VECTOR-SET! vect n expr)
Set the nth element of a vector (zero based).

1.21 Array Functions

(MAKE-ARRAY d1 d2...)

Make an array (vector of vectors) with the specified dimensions. At least one dimension must be specified.

(ARRAY-REF array s1 s2...)

Get an array element. The sn arguments are integer subscripts (zero based).

(ARRAY-SET! array s1 s2... expr)

Set an array element. The sn arguments are integer subscripts (zero based).

1.22 Table Functions

(MAKE-TABLE &optional size)

Make a table with the specified size. The size defaults to something useful hopefully.

(TABLE-REF table key)

Find the value in the table associated with the specified key.

(TABLE-SET! table key value)

Set the value in the table associated with the specified key.

(TABLE-REMOVE! table key)

Remove the entry with the specified key from the table. Return the old value associated with the key or nil if the key is not found.

(EMPTY-TABLE! table)

Remove all entries from a table.

(MAP-OVER-TABLE-ENTRIES table fun)

Apply the specified function to each entry in the table and return the list of values. The function should take two arguments. The first is the key and the second is the value associated with that key.

1.23 Conversion Functions

(SYMBOL->STRING sym)

Convert a symbol to a string. Returns the print name of the symbol as a string.

(STRING->SYMBOL str)

Convert a string to a symbol. Returns a symbol with the string as its print name. This can either be a new symbol or an existing one with the same print name.

(VECTOR->LIST vect)

Convert a vector to a list. Returns a list of the elements of the vector.

(LIST->VECTOR list)

Convert a list to a vector. Returns a vector of the elements of the list.

(STRING->LIST str)

Convert a string to a list. Returns a list of the characters in the string.

(LIST->STRING list)

Convert a list of character to a string. Returns a string whose characters are the elements of the list.

(CHAR->INTEGER char)

Convert a character to an integer. Returns the ASCII code of the character as an integer.

(INTEGER->CHAR n)

Convert an integer ASCII code to a character. Returns the character whose ASCII code is the integer.

(STRING->NUMBER str &optional base)

Convert a string to a number. Returns the value of the numeric interpretation of the string. The base argument must be 2, 8, 10 or 16 and defaults to 10.

(NUMBER->STRING n &optional base)

Convert a number to a string. Returns the string corresponding to the number. The base argument must be 2, 8, 10 or 16 and defaults to 10.

1.24 Logical Functions

(NOT expr)

Returns #t if the expression is #f and #t otherwise.

1.25 Type Predicates

(NULL? expr)

Returns #t if the expression is the empty list and #f otherwise.

(ATOM? expr)

Returns #f if the expression is a pair and #t otherwise.

(LIST? expr)

Returns #t if the expression is either a pair or the empty list and #f otherwise.

(NUMBER? expr)

Returns #t if the expression is a number and #f otherwise.

(BOOLEAN? expr)

Returns #t if the expression is either #t or #f and #f otherwise.

(PAIR? expr)

Returns #t if the expression is a pair and #f otherwise.

(SYMBOL? expr)

Returns #t if the expression is a symbol and #f otherwise.

(COMPLEX? expr)

Returns #t if the expression is a complex number and #f otherwise. Note: Complex numbers are not yet supported by XLISP.

(REAL? expr)

Returns #t if the expression is a real number and #f otherwise.

(RATIONAL? expr)

Returns #t if the expression is a rational number and #f otherwise. Note: Rational numbers are not yet supported by XLISP.

(INTEGER? expr)

Returns #t if the expression is an integer and #f otherwise.

(CHAR? expr)

Returns #t if the expression is a character and #f otherwise.

(STRING? expr)

Returns # if the expression is a string and #f otherwise.

(VECTOR? expr)

Returns #t if the expression is a vector and #f otherwise.

(TABLE? expr)

Returns #t if the expression is a table and #f otherwise.

(PROCEDURE? expr)

Returns #t if the expression is a procedure (closure) and #f otherwise.

(PORT? expr)

Returns #t if the expression is a port and #f otherwise.

(INPUT-PORT? expr)

Returns #t if the expression is an input port and #f otherwise.

(OUTPUT-PORT? expr)

Returns #t if the expression is an output port and #f otherwise.

(OBJECT? expr)

Returns #t if the expression is an object and #f otherwise.

(EOF-OBJECT? expr)

Returns #t if the expression is the object returned by READ upon detecting an end of file condition and #f otherwise.

(DEFAULT-OBJECT? expr)

Returns #t if the expression is the object passed as the default value of an optional parameter to a procedure when that parameter is omitted from a call and #f otherwise.

(ENVIRONMENT? expr)

Returns #t if the expression is an environment and #f otherwise.

1.26 Equality Predicates

(EQUAL? expr1 expr2)

Recursively compares two objects to determine if their components are the same and returns #t if they are the same and #f otherwise.

(EQV? expr1 expr2)

Compares two objects to determine if they are the same object. Returns #t if they are the same and #f otherwise. This function does not compare the elements of lists or vectors but will compare strings and all types of numbers.

(EQ? expr1 expr2)

Compares two objects to determine if they are the same object. Returns #t if they are the same and #f otherwise. This function performs a low level address compare on two objects and may return #f for objects that appear on the surface to be the same. This is because the objects are not stored uniquely in memory. For instance, numbers may appear to be equal, but EQ? will return #f when comparing them if they are stored at different addresses. The advantage of this function is that it is faster than the others. Symbols are guaranteed to compare correctly, so EQ? can safely be used to compare them.

1.27 Arithmetic Functions

(IDENTITY expr)

Returns the value of expr. This is the identity function.

(ZERO? n)

Returns #t if the number is zero and #f otherwise.

(POSITIVE? n)

Returns #t if the number is positive and #f otherwise.

(NEGATIVE? n)

Returns #t if the number is negative and #f otherwise.

(ODD? n)

Returns #t if the integer is odd and #f otherwise.

(EVEN? n)

Returns #t if the integer is even and #f otherwise.

(EXACT? n)

Returns #t if the number is exact and #f otherwise. Note: This function always returns #f in XLISP since exact numbers are not yet supported.

(INEXACT? n)

Returns #t if the number is inexact and #f otherwise. Note: This function always returns #t in XLISP since exact numbers are not yet supported.

(TRUNCATE n)

Truncates a number to an integer and returns the resulting value.

(FLOOR n)

Returns the largest integer not larger than n.

(CEILING n)

Returns the smallest integer not smaller than n.

(ROUND n)

Returns the closest integer to n, rounding to even when n is halfway between two integers.

(1+ n)

Returns the result of adding one to the number.

(-1+ n)

Returns the result of subtracting one from the number.

(ABS n)

Returns the absolute value of the number.

(GCD n1 n2...)

Returns the greatest common divisor of the specified numbers.

(LCM n1 n2...)

Returns the least common multiple of the specified numbers.

(RANDOM n)

Returns a random number between zero and n-1 (n must be an integer).

(SET-RANDOM-SEED! n)

Sets the seed of the random number generator to n.

(+ n1 n2...)

Returns the sum of the numbers.

(- n)

Negates the number and returns the resulting value.

(- n1 n2...)

Subtracts each remaining number from n1 and returns the resulting value.

(* n1 n2...)

Multiplies the numbers and returns the resulting value.

(/ n)

Returns 1/n.

(/ n1 n2...)

Divides n1 by each of the remaining numbers and returns the resulting value.

(QUOTIENT n1 n2...)

Divides the integer n1 by each of the remaining numbers and returns the resulting integer quotient. This function does integer division.

(REMAINDER n1 n2)

Divides the integer n1 by the integer n2 and returns the remainder.

(MODULO n1 n2)

Returns the integer n1 modulo the integer n2 .
(MIN n1 n2...)
Returns the number with the minimum value.
(MAX n1 n2...)
Returns the number with the maximum value.
(SIN n)
Returns the sine of the number.
(COS n)
Returns the cosine of the number.
(TAN n)
Returns the tangent of the number.
(ASIN n)
Returns the arc-sine of the number.
(ACOS n)
Returns the arc-cosine of the number.
(ATAN x)
Returns the arc-tangent of x.
(ATAN y x)
Returns the arc-tangent of y/x.
(EXP n)
Returns e raised to the n.
(SQRT n)
Returns the square root of n.
(EXPT n1 n2)
Returns n1 raised to the n2 power.
(LOG n)
Returns the natural logarithm of n.

1.28 Comparison Functions

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

These functions compare numbers and return #t if the numbers match the predicate and #f otherwise. For instance, (< x y z) will return #t if x is less than y and y is less than z.

1.29 Bitwise Logical Functions

(LOGAND n1 n2...)
Returns the bitwise AND of the integer arguments.

(LOGIOR n1 n2...)
Returns the bitwise inclusive OR of the integer arguments.
(LOGXOR n1 n2...)
Returns the bitwise exclusive OR of the integer arguments.
(LOGNOT n)
Returns the bitwise complement of n.
(ASH n shift)
Arithmetically shift n left by the specified number of places (or right if shift is negative)
(LSH n shift)
Logically shift n left by the specified number of places (or right if shift is negative).

1.30 String Functions

(MAKE-STRING size)
Makes a string of the specified size initialized to nulls.
(STRING-LENGTH str)
Returns the length of the string.
(STRING-NULL? str)
Returns #t if the string has a length of zero and #f otherwise.
(STRING-APPEND str1...)
Returns the result of appending the string arguments. If no arguments are supplied, it returns the null string.
(STRING-REF str n)
Returns the nth character in a string.
(STRING-SET! str n c)
Sets the nth character of the string to c.
(SUBSTRING str &optional start end)
Returns the substring of str starting at start and ending at end (integers). The range is inclusive of start and exclusive of end.
(STRING-UPCASE str &key start end)
Return a copy of the specified string with lowercase letters converted to uppercase letters in the specified range (which defaults to the whole string).
(STRING-DOWNCASE str &key start end)
Return a copy of the specified string with uppercase letters converted to lowercase letters in the specified range (which defaults to the whole string).
(STRING-UPCASE! str &key start end)
Like STRING-UPCASE but modifies the input string.
(STRING-DOWNCASE! str &key start end)

Like STRING-DOWNCASE but modifies the input string.

(STRING-TRIM bag str)

Return a string with characters that are in bag (which is also a string) removed from both the left and right ends.

(STRING-LEFT-TRIM bag str)

Return a string with characters that are in bag (which is also a string) removed from the left end.

(STRING-RIGHT-TRIM bag str)

Return a string with characters that are in bag (which is also a string) removed from right end.

(STRING-SEARCH str1 str2 &key start1 end1 start2 end2 from-end?)

Search for the specified substring of str1 in the specified substring of str2 and return the starting offset when a match is found or nil if no match is found.

(STRING-CI-SEARCH str1 str2 &key start1 end1 start2 end2 from-end?)

Like STRING-SEARCH but case insensitive.

1.31 String Comparison Functions

(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)

These functions compare strings and return #t if the strings match the predicate and #f otherwise. For instance, (STRING< x y) will return #t if x is less than y. Case is significant. #A does not match #a.

(STRING-CI<? str1 str2 &key start1 end1 start2 end2) (STRING-CI=? str1 str2 &key start1 end1 start2 end2) (STRING-CI>? str1 str2 &key start1 end1 start2 end2) (STRING-CI<=? str1 str2 &key start1 end1 start2 end2) (STRING-CI/=? str1 str2 &key start1 end1 start2 end2) (STRING-CI>=? str1 str2 &key start1 end1 start2 end2)

These functions compare strings and return #t if the strings match the predicate and #f otherwise. For instance, (STRING-CI< x y) will return #t if x is less than y. Case is not significant. #A matches #a.

1.32 Character Functions

(CHAR-UPPER-CASE? ch)

Is the specified character an upper case letter?

(CHAR-LOWER-CASE? ch)

Is the specifed character a lower case letter?

(CHAR-ALPHABETIC? ch)

Is the specified character an upper or lower case letter?

(CHAR-NUMERIC? ch)

Is the specified character a digit?

(CHAR-ALPHANUMERIC? ch)

Is the specified character a letter or a digit?

(CHAR-WHITESPACE? ch)

Is the specified character whitespace?

(STRING ch)

Return a string containing just the specified character.

(CHAR str [n])

Return the nth character of the string (n defaults to zero).

(CHAR-UPCASE ch)

Return the uppercase equivilent to the specified character if it is a letter. Otherwise, just return the character.

(CHAR-DOWNCASE ch)

Return the lowercase equivilent to the specified character if it is a letter. Otherwise, just return the character.

(DIGIT->CHAR n)

Return the character associated with the specified digit. The argument must be in the range of zero to nine.

1.33 Character Comparison Functions

(CHAR<? ch1 ch2) (CHAR=? ch1 ch2) (CHAR>? ch1 ch2) (CHAR<=? ch1 ch2) (CHAR/=? ch1 ch2) (CHAR>=? ch1 ch2)

These functions compare characters and return #t if the characters match the predicate and #f otherwise. For instance, (CHAR< x y) will return #t if x is less than y. Case is significant. #A does not match #a.

(CHAR-CI<? ch1 ch2) (CHAR-CI=? ch1 ch2) (CHAR-CI>? ch1 ch2) (CHAR-CI<=? ch1 ch2) (CHAR-CI>=? ch1 ch2)

These functions compare characters and return #t if the characters match the predicate and #f otherwise. For instance, (CHAR-CI< x y) will return #t if x is less than y. Case is not significant. #A matchs #a.

1.34 The Reader

(READ &optional port)

Reads an expression from the specified port. If no port is specified, the current input port is used. Returns the expression read or an object that satisfies the eof-object? predicate if it reaches the end of file on the port.

(READ-DELIMITED-LIST ch &optional port)

Read expressions building a list until the first occurance of the specified character. Return the resulting list.

(SET-MACRO-CHARACTER! ch fun &optional non-terminating? table)

(GET-MACRO-CHARACTER ch &optional table)

(MAKE-DISPATCH-MACRO-CHARACTER ch &optional non-terminating? table)

(SET-DISPATCH-MACRO-CHARACTER dch ch fun &optional table)

(GET-DISPATCH-MACRO-CHARACTER dch ch &optional table)

1.35 The Printer

(WRITE expr &optional port)

Writes an expression to the specified port. If no port is specified, the current output port is used. The expression is written such that the READ function can read it back. This means that strings will be enclosed in quotes and characters will be printed with # notation.

(WRITE-SIZE expr)

Returns the number of characters in the printed representation of the specified object when printed by the function WRITE.

(DISPLAY-SIZE expr)

Returns the number of characters in the printed representation of the specified object when printed by the function DISPLAY.

(DISPLAY expr &optional port)

Writes an expression to the specified port. If no port is specified, the current output port is used. The expression is written without any quoting characters. No quotes will appear around strings and characters are written without the # notation.

(PRINT expr &optional port)

The same as (NEWLINE port) followed by (WRITE expr port).

1.36 Input/Output Functions

(READ-LINE &optional port)

Read a line from the specified port (which defaults to the current input port). Returns the line read as a string or nil if it reaches end of file on the port.

(READ-CHAR &optional port)

Reads a character from the specified port. If no port is specified, the current input port is used. Returns the character read or an object that satisfies the default-object? predicate if it reaches the end of file on the port.

(UNREAD-CHAR ch &optional port)

Unread the specified character. This causes it to be the next character read from the port. Only one character can be "unread" at a time. This allows for a one character look ahead for parsers.

(PEEK-CHAR &optional port)

Peek at the next character without actually reading it.

(CHAR-READY? &optional port)

Returns #t if a character is ready on the specified port, #f if not.

(CLEAR-INPUT &optional port)

Clears any buffered input on the specified port.

(READ-BYTE &optional port)

Reads a byte from the specified port. If no port is specified, the current input port is used. Returns the byte read or an object that satisfies the default- object? predicate if it reaches the end of file on the port.

(READ-SHORT &optional port) (READ-SHORT-HIGH-FIRST &optional port) (READ-SHORT-LOW-FIRST &optional port)

Read signed 16 bit value from the specified port in whatever byte order is native to the host machine. Returns the 16 bit value or an object that satisfies the eof-object? predicate if it reaches the end of file on the port. The -HIGH-FIRST and -LOW-FIRST forms read the high and low byte first respectively.

(READ-LONG &optional port) (READ-LONG-HIGH-FIRST &optional port) (READ-LONG-LOW-FIRST &optional port)

Read signed 32 bit value from the specified port in whatever byte order is native to the host machine. Returns the 32 bit value or an object that satisfies the eof-object? predicate if it reaches the end of file on the port. . The -HIGH-FIRST and -LOW-FIRST forms read the high and low byte first respectively.

(WRITE-CHAR ch &optional port)

Writes a character to the specified port. If no port is specified, the current output port is used.

(WRITE-BYTE ch &optional port)

Writes a byte to the specified port. If no port is specified, the current output port is used.

(WRITE-SHORT n &optional port) (WRITE-SHORT-HIGH-FIRST n &optional port) (WRITE-SHORT-LOW-FIRST n &optional port)

Write a signed 16 bit integer to the specified port. If no port is specified, the current output port is used. The -HIGH-FIRST and -LOW-FIRST forms write the high and low byte first respectively.

(WRITE-LONG n &optional port) (WRITE-LONG-HIGH-FIRST n &optional port) (WRITE-LONG-LOW-FIRST n &optional port)

Write a signed 32 bit integer to the specified port. If no port is specified, the current output port is used. . The -HIGH-FIRST and -LOW-FIRST forms write the high and low byte first respectively.

(NEWLINE &optional port)

Starts a new line on the specified port. If no port is specified, the current output port is used.

(FRESH-LINE &optional port)

Starts a fresh line on the specified port. If the output position is already at the start of the line, FRESH-LINE does nothing. If no port is specified, the current output port is used.

1.37 Format

(FORMAT port str &rest args)

If port is #f, FORMAT collects its output into a string and returns the string. If port is #t, FORMAT sends its output to the current output port. Otherwise, port should be an output port.

~S print argument as if with WRITE

~A print argument as if with DISPLAY

~X print argument as a hexadecimal number

~% print as if with NEWLINE

~& print as if with FRESH-LINE

1.38 Output Control Functions

(PRINT-BREADTH [n])

Controls the maximum number of elements of a list that will be printed. If n is an integer, the maximum number is set to n. If it is #f, the limit is set to infinity. This is the default. If n is omitted from the call, the current value is returned.

(PRINT-DEPTH [n])

Controls the maximum number of levels of a nested list that will be printed. If n is an integer, the maximum number is set to n. If it is #f, the limit is set to infinity. This is the default. If n is omitted from the call, the current value is returned.

1.39 File I/O Functions

All four of the following OPEN functions take an optional argument to indicate that file I/O is to be done in binary mode. For binary files, this argument should be the symbol BINARY. For text files, the argument can be left out or the symbol TEXT can be supplied.

(OPEN-INPUT-FILE str ['binary])

Opens the file named by the string and returns an input port.

(OPEN-OUTPUT-FILE str ['binary])

Creates the file named by the string and returns an output port.

(OPEN-APPEND-FILE str ['binary])

Opens the file named by the string for appending returns an output port.

(OPEN-UPDATE-FILE str ['binary])

Opens the file named by the string for input and output and returns an input/output port.

(FILE-MODIFICATION-TIME str)

Returns the time the file named by the string was last modified.

(PARSE-PATH-STRING str)

Parses a path string and returns a list containing each path entry terminated by a path separator.

(GET-FILE-POSITION port)

Returns the current file position as an offset in bytes from the beginning of the file.

(SET-FILE-POSITION! port offset whence)

Sets the current file position as an offset in bytes from the beginning of the file (when whence equals 0), the current file position (when whence equals 1) or the end of the file (when whence equals 2). Returns the new file position as an offset from the start of the file.

(CLOSE-PORT port)

Closes any port.

(CLOSE-INPUT-PORT port)

Closes an input port.

(CLOSE-OUTPUT-PORT port)

Closes an output port.

(CALL-WITH-INPUT-FILE str proc)

Open the file whose name is specifed by str and call proc passing the resulting input port as an argument. When proc returns, close the file and return the value returned by proc as the result.

(CALL-WITH-OUTPUT-FILE str proc)

Create the file whose name is specifed by str and call proc passing the resulting output port as an argument. When proc returns, close the file and return the value returned by proc as the result.

(CURRENT-INPUT-PORT)

Returns the current input port.

(CURRENT-OUTPUT-PORT)

Returns the current output port.

(CURRENT-ERROR-PORT)

Returns the current error port.

1.40 String Stream Functions

(MAKE-STRING-INPUT-STREAM str)

Make a stream that can be used to retrieve the characters in the specified string. The returned stream can be used as an input port in any function that takes an input port as an argument.

(MAKE-STRING-OUTPUT-STREAM)

Make a stream that can be used as an output port in any function that takes an output port as an argument. The stream accumulates characters until the GET-OUTPUT-STREAM-STRING function is called to retrieve them.

(GET-OUTPUT-STREAM-STRING stream)

Returns the contents of a string output stream as a string and clears the output stream.

1.41 Control Features

(EVAL expr [env])

Evaluate the expression in the global environment and return its value.

(APPLY proc args)

Apply the procedure to the list of arguments and return the result.

(MAP proc list...)

Apply the procedure to argument lists formed by taking corresponding elements from each list. Form a list from the resulting values and return that list as the result of the MAP call.

(FOR-EACH fun list...)

Apply the procedure to argument lists formed by taking corresponding elements from each list. The values returned by the procedure applications are discarded. The value returned by FOR-EACH is unspecified.

(CALL-WITH-CURRENT-CONTINUATION proc) (CALL/CC proc)

Form an "escape procedure" from the current continuation and pass it as an argument to proc. Calling the escape procedure with a single argument will cause that argument to be passed to the continuation that was in effect when the CALL-WITH-CURRENT-CONTINUATION procedure was called.

1.42 Environment Functions

(THE-ENVIRONMENT)

Returns the current environment.

(PROCEDURE-ENVIRONMENT proc)

Returns the environment from a procedure closure.

(ENVIRONMENT-BINDINGS env)

Returns an association list corresponding to the top most frame of the specified environment.

(ENVIRONMENT-PARENT env)

Returns the parent environment of the specified environment.

(BOUND? symbol [env])

Returns #t if the symbol is bound in the environment.

(SYMBOL-VALUE symbol [env])

Returns the value of a variable in an environment.

(SET-SYMBOL-VALUE! symbol value [env])

Sets the value of a symbol in an environment. The result of the set-symbol-value! expression is unspecified.

(EVAL expr [env])

Evaluate the expression in the specified environment and return its value.

1.43 Utility Functions

(LOAD str)

Read and evaluate each expression from the specified file.

(LOAD-NOISILY str)

Read and evaluate each expression from the specified file and print the results to the current output port.

(LOAD-FASL-FILE str)

Load a fasl file produced by COMPILE-FILE.

(TRANSCRIPT-ON str)

Opens a transcript file with the specified name and begins logging the interactive session to that file.

(TRANSCRIPT-OFF)

Closes the current transcript file.

(COMPILE expr &optional env)

Compiles an expression in the specified environment and returns a thunk that when called causes the expression to be evaluated. The environment defaults to the top level environment if not specified.

(SAVE name)

Saves the current workspace to a file with the specified name. The workspace can later be reloaded using RESTORE.

(RESTORE name)

Restores a previously saved workspace from the file with the specified name.

(GETARG n)

Get the nth argument from the command line. If there were fewer than n arguments, return nil.

(GET-TIME)

Get the current time in seconds.

(GET-ENVIRONMENT-VARIABLE name)

Get the value of the environment variable with the specified name. The name should be a string. Returns the value of the environment variable if it exists. Otherwise, returns nil.

(EXIT) (QUIT)

Exits from XLISP back to the operating system.

(GC [ni vi])

Invokes the garbage collector and returns information on memory usage. If ni and vi are specified, they must be integers. Node and vector space are expanded by those amounts respectively and no garbage collection is triggered. GC returns an array of six values: the number of calls to the garbage collector, the total number of nodes, the current number of free nodes, the number of node segments, the number of vector segments and the total number of bytes allocated to the heap.

(ROOM)

Returns the same information as GC without actually invoking the garbage collector.

1.44 Fast Loading

(LOAD-FASL-FILE name)
(FASL-WRITE-PROCEDURE proc &optional port)
(FASL-READ-PROCEDURE &optional port)

1.45 C Records

(DEFINE-CRECORD name (field-definition...))
Field definition:
(field-name type &optional size)
Where type is:
char, uchar, short, ushort, int, uint, long, ulong, str ** (ALLOCATE-CMEMORY type size)
(FREE-CMEMORY ptr)
(FOREIGN-POINTER? ptr)
(FOREIGN-POINTER-TYPE ptr)
(SET-FOREIGN-POINTER-TYPE! ptr type)
(FOREIGN-POINTER-TYPE? ptr type)
(FOREIGN-POINTER-EQ? ptr1 ptr2)
(GET-CRECORD-FIELD ptr offset type)
(GET-CRECORD-FIELD-ADDRESS ptr offset type)
(SET-CRECORD-FIELD! ptr offset type val)
(GET-CRECORD-STRING ptr offset length)
(SET-CRECORD-STRING! ptr offset length str)
(GET-CRECORD-TYPE-SIZE type) * *

1.46 Debugging Functions

(DECOMPILE proc &optional port)
Decompiles the specified bytecode procedure and displays the bytecode instructions to the specified port. If not specified, the port defaults to the current output port.
(INSTRUCTION-TRACE &rest body)
Enables bytecode level instruction tracing during the evaluation of the expressions in the body.
(TRACE-ON)
Starts bytecode instruction level tracing.
(TRACE-OFF)
Stops bytecode instruction level tracing.
(SHOW-STACK &optional n)

Shows the call stack leading up to an error when invoked from a debug prompt. Each line represents a procedure waiting for a value. The line is displayed in the form of a function call with the procedure first followed by the actual values of the arguments that were passed to the procedure. For method invocations, the method is first followed by the object receiving the message followed by the arguments. N is the number of stack levels to display. If unspecified, it defaults to 20.

(SHOW-CONTROL-STACK &optional n)

Shows frames on the continuation stack. N is the number of stack levels to display. If unspecified, it defaults to 20.

(SHOW-VALUE-STACK &optional n)

Shows frames on the value stack. N is the number of stack levels to display. If unspecified, it defaults to 20.

(RESET)

Returns to the top level read/eval/print loop.

1.47 System Functions

(%CAR pair) (%CDR pair) (%SET-CAR! pair expr) (%SET-CDR! pair expr) (%VECTOR-LENGTH vect) (%VECTOR-REF vect n) (%VECTOR-SET! vect n expr)

These functions do the same as their counterparts without the leading '%' character. The difference is that they don't check the type of their first argument. This makes it possible to examine data structures that have the same internal representation as pairs and vectors. It is **very** dangerous to modify objects using these functions and there is no guarantee that future releases of XLISP will represent objects in the same way that the current version does.

(%VECTOR-BASE vect)

Returns the address of the base of the vector storage.

(%ADDRESS-OF expr)

Returns the address of the specified object in the heap.

(%FORMAT-ADDRESS addr)

Returns the address of an object as a string formated for output as a hex number.

1.48 Object Representations

This version of XLISP uses the following object representations:

Closures are represented as pairs. The car of the pair is the compiled function and the cdr of the pair is the saved environment.

Compiled functions are represented as vectors. The element at offset 0 is the bytecode string. The element at offset 1 is the function name. The element at offset 2 is a list of names of the function arguments. The elements at offsets above 2 are the literals refered to by the compiled bytecodes.

Environments are represented as lists of vectors. Each vector is an environment frame. The element at offset 0 is a list of the symbols that are bound in that frame. The symbol values start at offset 1.

Objects are represented as vectors. The element at offset 0 is the class of the object. The remaining elements are the object's instance variable values.