XLISP: An Object-oriented Lisp

Version 2.0

February 6, 1988

by
David Michael Betz
P.O. Box 144
Peterborough, NH 03458

(603) 924-4145 (home)

Additions to the manual in *italics*, by Tom Almy October 26, 1990.

This distributed version has the added functions of XLISP 2.1 incorporated, as well as other enhancements, all of which can be eliminated via compilation options.

Table of Contents

INTRODUCTION

XLISP is an experimental programming language combining some of the features of Common Lisp with an object-oriented extension capability. It was implemented to allow experimentation with object-oriented programming on small computers.

There are currently implementations of XLISP running on the IBM-PC and clones under MS-DOS, on the Macintosh, the Atari-ST and 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 to non-commercial users.

Many Common 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.

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

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

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 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 the workspace "xlisp.wks" from the current directory. If that file doesn't exist, *or the "-w" flag is in the command line,* XLISP builds an initial workspace, empty except for the built-in functions and symbols.

Then*, providing xlisp.wks was not loaded,* XLISP attempts to load "init.lsp" from the current directory. It then loads any files named as parameters on the command line (after appending ".lsp" to their names). *If the "-v" flag is in the command line, then the files are loaded verbosely. The option "-t filename" will open a transcript file of the name "filename".*

XLISP then issues the following prompt:

>

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

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

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

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

The following control characters can be typed while XLISP is executing:

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

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

BREAK COMMAND LOOP

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

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

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

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

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

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

DATA TYPES

(Major modifications)

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

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

- lists
  NIL is a special pointer.
- arrays
  The CDR field of an array points to the dynamically allocated data array, while the CAR contains the integer length of the array. Elements in the data array are pointers to other cells [Size limited to 16383]
- character strings
  Implemented like arrays, except string array is byte indexed and contains the actual characters. Note that unlike the underlying C, the null character (value 0) is valid. [Size limited to 32767]
- symbols
  Implemented as a 4 element array. The elements are value cell, function cell, property list, and print name (a character string node).
- integers
  Small integers (> -129 and <256) are statically allocated and are thus always EQ integers of the same value. The CAR field is used to hold the value, which is a 32 bit signed integer.
- characters
  All characters are statically allocated and are thus EQ characters of the same value. The CAR field is used to hold the value. In XLISP characters are "unsigned" and thus range in value from 0 to 255.
- floats
  The CAR and CDR fields hold the value, which is typically a 64 bit IEEE floating point number.
- objects
  Implemented as an array of instance variable count plus one elements. The first element is the object’s class, while the remaining arguments are the instance variables.

- streams (file)
  The CAR and CDR fields are used in a system dependent way as a file pointer. Files are not kept open across image saves, but the standard files are opened automatically during image restores.
- streams (string)
  Implemented as a TCONC list of characters (see page 34).
- subrs (built-in functions)
  The CAR field points to the actual code to execute, while the CDR field is an internal pointer to the name of the function.
- fsubrs (special forms)
  Same implementation as subrs.
- closures (user defined functions)
  Implemented as an array of 11 elements:
  1. name symbol or NIL
  2. ’lambda or ’macro
  3. list of required arguments
  4. optional arguments as list of (<arg> <init> <specified-p>) triples.
  5. &rest argument
  6. &key arguments as list of (<key> <arg> <init> <specified-p>) quadruples.
  7. &aux arguments as list of (<arg> <init>) pairs.
  8. function body
  9. value environment (see page 64 for format)
  10. function environment
  11. argument list (unprocessed)
- structures
  Implemented as an array with first element being a pointer to the structure name string, and the remaining elements being the structure elements.

THE EVALUATOR

The process of evaluation in XLISP:

Strings, integers, characters, floats, objects, arrays, *structures,* streams, subrs, fsubrs and closures evaluate to themselves.

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

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

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

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

> If it is a subr or closure, the remaining list elements are evaluated and the subr or closure is applied to these evaluated expressions.
>
> If it is an fsubr, the fsubr is called with the remaining list elements as arguments (unevaluated).
>
> If it is a macro, the macro is expanded with the remaining list elements as arguments (unevaluated). The macro expansion is then evaluated in place of the original macro call.

HOOK FUNCTIONS

(New section)

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

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

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

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

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

LEXICAL CONVENTIONS

(Major expansion of original document)

The following conventions must be followed when entering XLISP programs:

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

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

( ) ' ` , " ;

and the escape characters:

\ |

In addition, the first character may not be '#' (non-terminating macro character), nor may the symbol have identical syntax with an integer or floating point literal. Uppercase and lowercase characters are not distinguished within symbol names. All lowercase characters are mapped to uppercase on input.

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

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

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

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

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

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

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

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

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

READTABLES

(Major modifications)

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

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

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

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

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

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

XLISP defines several useful read macros:

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

LAMBDA LISTS

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

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

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

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

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

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

Here is the complete syntax for lambda lists:

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

where:

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

OBJECTS

Definitions:

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

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

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

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

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

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

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

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

THE 'Object' CLASS

Object THE TOP OF THE CLASS HEIRARCHY

Messages:

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

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

:prin1 [<stream>] PRINT THE OBJECT
returns the object

:isnew THE DEFAULT OBJECT INITIALIZATION ROUTINE
returns the object

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

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

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

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

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

THE 'Class' CLASS

Class THE CLASS OF ALL OBJECT CLASSES (including itself)

Messages:

:new — CREATE A NEW INSTANCE OF A CLASS

returns	the new class object

:isnew <ivars> [<cvars> [<super>]] — INITIALIZE A NEW CLASS

<ivars>	the list of instance variable symbol
<cvars>	the list of class variable symbols
<super>	the superclass (default is Object)
returns	the new class object

:answer <msg> <fargs> <code> — ADD A MESSAGE TO A CLASS

<msg>	the message symbol
<fargs>	the formal argument list (lambda list)
<code>	a list of executable expressions
returns	the object

:superclass — *GET THE SUPERCLASS OF THE OBJECT*

returns	*the superclass (of the class)*

(defined in classes.lsp)

:messages — *GET THE LIST OF MESSAGES OF THE CLASS*

returns	*association list of message selectors and closures for messages.*

(defined in classes.lsp)

:storeon — *READ REPRESENTATION*

returns	*a list, that when executed will re-create the class and its methods.*

(defined in classes.lsp)

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

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

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

INSTANCE VARIABLES OF CLASS 'CLASS':

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

IVARS - List of names of instance variables.

CVARS - List of names of class variables.

CVAL - List of class variable values.

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

IVARCNT - instance variables in this class (length of IVARS)

IVARTOTAL - total instance variables for this class and all superclasses of this class.

PNAME - printname string for this class.

SYMBOLS

- *nil - the empty list*
- *t - truth value*
- self - the current object (within a method context)
- *object - the class 'Object'*
- *class - the class 'Class'*
- *obarray* - the object hash table
- *standard-input* - the standard input stream*, stdin*
- *standard-output* - the standard output stream*, stdout*
- *error-output* - the error output stream*, stderr*
- *trace-output* - the trace output stream*, defaults to stderr*
- *debug-io* - the break loop i/o stream*, defaults to stderr*
- *breakenable* - flag controlling entering break loop on errors
- *tracelist* - list of names of functions to trace
- *tracenable* - enable trace back printout on errors
- *tracelimit* - number of levels of trace back information
- *evalhook* - user substitute for the evaluator function
- *applyhook* - *user substitute for function application*
- *readtable* - the current readtable
- *unbound* - indicator for unbound symbols
- *gc-flag* - controls the printing of gc messages
- *gc-hook* - function to call after garbage collection
- *integer-format* - format for printing integers ("%d" or "%ld")
- *float-format* - format for printing floats ("%g")
- *print-case* - symbol output case (:upcase or :downcase)
- **print-level* - list levels beyond this setting are printed as '#'*
- **print-length* - lists longer than this setting are printed as '...'*
- **dos-input* - use dos line input function for read.*

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

Intended to be added:

- **print-base* - radix used in printing integers.*

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). *May not be macro or fsubr.*
<args> the argument list *(unlike Common Lisp, only one allowed)*
returns the result of applying the function to the arguments

(funcall <fun> <arg>...) CALL A FUNCTION WITH ARGUMENTS

<fun> the function to call (or function symbol). *May not be macro or fsubr.*
<arg> arguments to pass to the function
returns the result of calling the function with the arguments

(quote <expr>) RETURN AN EXPRESSION UNEVALUATED

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

(function <expr>) GET THE FUNCTIONAL INTERPRETATION

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

(backquote <expr>) FILL IN A TEMPLATE

fsubr
<expr> the template (quoted)
returns a copy of the template with comma and comma-at expressions expanded.

(comma <expr>) COMMA EXPRESSION

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

(comma-at <expr>) COMMA-AT EXPRESSION

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

(lambda <args> <expr>...) MAKE A FUNCTION CLOSURE

fsubr
<args> formal argument list (lambda list) (quoted)
<expr> expressions of the function body (quoted)
returns the function closure

(get-lambda-expression <closure>) GET THE LAMBDA EXPRESSION

<closure>	the closure
returns	the original lambda expression

(macroexpand <form>) RECURSIVELY EXPAND MACRO CALLS

<form>	the form to expand
returns	the macro expansion

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

<form>	the macro call form
returns	the macro expansion

SYMBOL FUNCTIONS

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

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

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

fsubr

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

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

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

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

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

fsubr

<place>	the field specifier (quoted):	
	<sym>	set value of a symbol
	(car <expr>)	set car of a cons node
	(cdr <expr>)	set cdr of a cons node
	(nth <n> <expr>)	set nth car of a list
	(aref <expr> <n>)	set nth element of an array *or string*
	(elt <expr> <n>)	*set nth element of a sequence*
	(get <sym> <prop>)	set value of a property
	(symbol-value <sym>)	set value of a symbol
	(symbol-function <sym>)	functional value of a symbol
	(symbol-plist <sym>)	set property list of a symbol
	(send <obj> :<ivar>)	*(When classes.lsp used), set instance variable of object.*
	(<sym>-<element> <struct>)	*(When struct.lsp used), set the element of structure struct, type sym.*
	(<fieldsym> <args>)	*the function stored in property *setf* in symbol <fieldsym> is applied to (<args> <expr>)*
<value>	the new value	
returns	the new value	

(push <expr> <place>) *CONS TO SYMBOL VALUE*
<place> *field specifier being modified (see setf)*
<expr> *value to cons to current symbol value*
returns *the new value which is (CONS <expr> <place>)*
Note: defined as macro in common.lsp

(pop <place>) *REMOVE FIRST ELEMENT OF SYMBOL VALUE*
<place> *the field being modified (see setf)*
returns *(CAR <place>), field changed to (CDR <place>)*
Note: defined as macro in common.lsp

(defun <sym> <fargs> <expr>...) DEFINE A FUNCTION
(defmacro <sym> <fargs> <expr>...) DEFINE A MACRO
fsubr
<sym> symbol being defined (quoted)
<fargs> formal argument list (lambda list) (quoted)
<expr> expressions constituting the body of the function (quoted)
returns the function symbol

(gensym [<tag>]) GENERATE A SYMBOL
<tag> string or number
returns the new symbol

(intern <pname>) MAKE AN INTERNED SYMBOL
<pname> the symbol's print name string
returns the new symbol

(make-symbol <pname>) MAKE AN UNINTERNED SYMBOL
<pname> the symbol's print name string
returns the new symbol

(symbol-name <sym>) GET THE PRINT NAME OF A SYMBOL
<sym> the symbol
returns the symbol's print name

(symbol-value <sym>) GET THE VALUE OF A SYMBOL
<sym> the symbol
returns the symbol's value

(symbol-function <sym>) GET THE FUNCTIONAL VALUE OF A SYMBOL
<sym> the symbol
returns the symbol's functional value

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

(hash <sym> <n>) COMPUTE THE HASH INDEX FOR A SYMBOL

<sym>	the symbol or string
<n>	the table size (integer)
returns	the hash index (integer)

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

<sym>	*the symbol*
returns	*the symbol*

Note: defined in init.lsp

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

<sym>	*the symbol*
returns	*the symbol*

Note: defined in init.lsp

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

This function now also works on strings.

<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

(vector <expr>...) MAKE AN INITIALIZED VECTOR

<expr>	the vector elements
returns	the new vector

SEQUENCE FUNCTIONS

These functions work on sequences -- lists, arrays, or strings, and for the most part represent an extension of the distribution XLISP 2.0.

(concatenate <type> <expr> ...) CONCATENATE SEQUENCES

<type>	result type, one of cons, array, or string
<expr>	zero or more sequences to concatenate
returns	a sequence which is the concatenation of the arguement sequences

NOTE: if result type is string, sequences must contain only characters.

(elt <expr> <n>) GET THE NTH ELEMENT OF A SEQUENCE

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

(map <type> <fcn> <expr> ...) APPLY FUNCTION TO SUCCESSIVE ELEMENTS

<type>	result type, one of cons, array, string, or nil
<fcn>	the function or function name
<expr>	a sequence for each argument of the function
returns	a new sequence of type <type>.

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

<fcn>	the function or function name
<expr>	a sequence for each argument of the function
returns	every returns last evaluated function result, or T
	notevery returns T if there is a nil function result, else NIL

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

<fcn>	the function or function name
<expr>	a sequence for each argument of the function
returns	some returns first non-nil function result, or NIL
	notany returns NIL if there is a non-nil function result, else T

(length <expr>) FIND THE LENGTH OF A SEQUENCE

Unchanged from the distribution

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

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

Distribution only worked on lists.

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

(subseq <seq> <start> [<end>]) EXTRACT A SUBSEQUENCE

Distribution only worked on strings.

<seq> the sequence
<start> the starting position (zero origin)
<end> the ending position + 1 (defaults to end) *or NIL for end of sequence*
returns the sequence between <start> and <end>

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

Note: added function

<seq1> the sequence to search for
<seq2> the sequence to search in
:test the test function (defaults to eql)
:test-not the test function (sense inverted)
:start1 starting index in <seq1>
:end1 index of end+1 in <seq1> or NIL for end of sequence
:start2 starting index in <seq2>
:end2 index of end+1 in <seq2> or NIL for end of sequence
returns position of first match

(remove <expr> <seq> &key :test :test-not :start :end)
REMOVE ELEMENTS FROM A SEQUENCE

Distribution only worked on lists.

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

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

Distribution only worked on lists.

<test> the test predicate
<seq> the sequence
:start starting index
:end index of end+1, or NIL for (length <seq>)
returns copy of sequence with matching elements removed

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

Distribution only worked on lists.

<test>	the test predicate
<seq>	the sequence
:start	*starting index*
:end	*index of end+1, or NIL for (length <seq>)*
returns	copy of sequence with non-matching elements removed

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

Note: added function

<test>	the test predicate
<seq>	the sequence
:start	starting index
:end	index of end+1, or NIL for (length <seq>)
returns	count of matching elements

(find-if <test> <seq> &key :start :end)
FIND FIRST ELEMENT THAT PASSES TEST

Note: added function

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

(position-if <test> <seq> &key :start :end)
FIND POSITION OF FIRST ELEMENT THAT PASSES TEST

Note: added function

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

(delete <expr> <seq> &key :test :test-not :start :end)
DELETE ELEMENTS FROM A SEQUENCE

Distribution only worked on lists.

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

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

DELETE ELEMENTS THAT PASS TEST

Distribution only worked on lists.

<test>	the test predicate
<seq>	the sequence
:start	*starting index*
:end	*index of end+1, or NIL for (length <expr>)*
returns	the sequence with matching elements deleted

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

DELETE ELEMENTS THAT FAIL TEST

Distribution only worked on lists.

<test>	the test predicate
<seq>	the sequence
:start	*starting index*
:end	*index of end+1, or NIL for (length <expr>)*
returns	the sequence with non-matching elements deleted

LIST FUNCTIONS

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

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

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

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

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

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

(cons <expr1> <expr2>) CONSTRUCT A NEW LIST NODE

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

(list <expr>...) CREATE A LIST OF VALUES
(list <expr> ... <list>)*
list is defined in common.lsp*

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

(append <expr>...) APPEND LISTS

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

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

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

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

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

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

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

(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

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

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

Note: this function is in init.lsp

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

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

Note: this function is in init.lsp

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

Modified to do minimum copying as in Common Lisp

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

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

Modified to do minimum copying as in Common Lisp

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

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

In file common.lsp

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

(copy-list <list>) *COPY THE TOP LEVEL OF A LIST*

In file common.lsp

<list>	*the list*
returns	*a copy of the list (new cons cells in top level)*

(copy-alist <alist>) *COPY AN ASSOCIATION LIST*

In file common.lsp

<alist>	*the association list*
returns	*a copy of the association list (keys and values not copies)*

(copy-tree <tree>) *COPY A TREE*

In file common.lsp

<tree>	*a tree structure of cons cells*
returns	*a copy of the tree structure*

(make_tconc) *MAKE A TCONC STRUCTURE*

In file common.lsp. (note that string streams are internally implemented as TCONCs. Older versions of xlisp allowed any TCONC to be used as a steam, but 2.0 implements a special string stream type.)

returns	*an empty tconc structure*

(tconc <tconc> <expr>) *ADD TO TAIL OF TCONC STRUCTURE*
(lconc <tconc> <list>)

In file common.lsp

<tconc>	*a tconc structure*
<expr>	*element to add to tail*
<list>	*list of elements to add to tail*
returns	*modified tconc*

(remove-head <tconc>) *REMOVE FROM HEAD OF TCONC STRUCTURE*

In file common.lsp

<tconc>	*a tconc structure*
returns	*head of tconc (tconc is modified)*

DESTRUCTIVE LIST FUNCTIONS

See also nreverse, delete, delete-if, and delete-if-not, under SEQUENCE FUNCTIONS.

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

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

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

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

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

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

(sort <list> <test>) SORT A LIST

<list>	the list to sort
<test>	the comparison function
returns	the sorted list

ARITHMETIC FUNCTIONS

Warning: integer calculations that overflow give erroneous results.
ASIN, ACOS, and ATAN are retrofitted from XLISP 2.1.

(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 OR INVERT A SINGLE NUMBER
<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

(gcd <n1> <n2>...) COMPUTE THE GREATEST COMMON DIVISOR

<n1>	the first number (integer)
<n2>	the second number(s) (integer)
returns	the greatest common divisor

(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

(asin <expr>) COMPUTE THE ARC SINE OF A NUMBER

<expr>	the floating point number
returns	the arc sine of the number

(acos <expr>) COMPUTE THE ARC COSINE OF A NUMBER

<expr>	the floating point number
returns	the arc cosine of the number

(atan <expr>) COMPUTE THE ARC TANGENT OF A NUMBER

<expr>	the floating point number
returns	the arc 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

(< <n1> <n2>...) TEST FOR LESS THAN
(<= <n1> <n2>...) TEST FOR LESS THAN OR EQUAL TO
(= <n1> <n2>...) TEST FOR EQUAL TO
(/= <n1> <n2>...) TEST FOR NOT EQUAL TO
(>= <n1> <n2>...) TEST FOR GREATER THAN OR EQUAL TO
(> <n1> <n2>...) TEST FOR GREATER THAN
<n1> the first number to compare
<n2> the second number to compare
returns the result of comparing <n1> with <n2>...

BITWISE LOGICAL FUNCTIONS

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

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

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

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

STRING FUNCTIONS

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

(string <expr>) MAKE A STRING FROM AN INTEGER ASCII VALUE
- <expr> — *an integer (which is first converted into its ASCII character value), string, character, or symbol*
- returns — *the string representation of the argument*

(string-trim <bag> <str>) TRIM BOTH ENDS OF A STRING
- <bag> — a string containing characters to trim
- <str> — the string to trim
- returns — a trimed copy of the string

(string-left-trim <bag> <str>) TRIM THE LEFT END OF A STRING
- <bag> — a string containing characters to trim
- <str> — the string to trim
- returns — a trimed copy of the string

(string-right-trim <bag> <str>) TRIM THE RIGHT END OF A STRING
- <bag> — a string containing characters to trim
- <str> — the string to trim
- returns — a trimed copy of the string

(string-upcase <str> &key :start :end) CONVERT TO UPPERCASE
- <str> — the string
- :start — the starting offset
- :end — the ending offset + 1 *or NIL for end of string*
- returns — a converted copy of the string

(string-downcase <str> &key :start :end) CONVERT TO LOWERCASE
- <str> — the string
- :start — the starting offset
- :end — the ending offset + 1 *or NIL for end of string*
- returns — a converted copy of the string

(nstring-upcase <str> &key :start :end) CONVERT TO UPPERCASE
- <str> — the string
- :start — the starting offset
- :end — the ending offset + 1 *or NIL for end of string*
- returns — the converted string (not a copy)

(nstring-downcase <str> &key :start :end) CONVERT TO LOWERCASE

<str>	the string
:start	the starting offset
:end	the ending offset + 1 *or NIL for end of string*
returns	the converted string (not a copy)

(strcat <expr>...) CONCATENATE STRINGS

Macro in init.lsp, to maintain compatibility with distribution.
See "concatenate".

<expr>	the strings to concatenate
returns	the result of concatenating the strings

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

<str1>	the first string to compare
<str2>	the second string to compare
:start1	first substring starting offset
:end1	first substring ending offset + 1 *or NIL for end of string*
:start2	second substring starting offset
:end2	second substring ending offset + 1 *or NIL for end of string*
returns	t if predicate is true, nil otherwise

Note: case is significant with these comparison functions.

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

<str1>	the first string to compare
<str2>	the second string to compare
:start1	first substring starting offset
:end1	first substring ending offset + 1 *or NIL for end of string*
:start2	second substring starting offset
:end2	second substring ending offset + 1 *or NIL for end of string*
returns	t if predicate is true, nil otherwise

Note: case is not significant with these comparison functions.

CHARACTER FUNCTIONS

(char <string> <index>)	EXTRACT A CHARACTER FROM A STRING
<string>	the string
<index>	the string index (zero relative)
returns	the ascii code of the character

(upper-case-p <chr>)	IS THIS AN UPPER CASE CHARACTER?
<chr>	the character
returns	true if the character is upper case, nil otherwise

(lower-case-p <chr>)	IS THIS A LOWER CASE CHARACTER?
<chr>	the character
returns	true if the character is lower case, nil otherwise

(both-case-p <chr>)	IS THIS AN ALPHABETIC (EITHER CASE) CHARACTER?
<chr>	the character
returns	true if the character is alphabetic, nil otherwise

(digit-char-p <chr>)	IS THIS A DIGIT CHARACTER?
<chr>	the character
returns	the digit weight if character is a digit, nil otherwise

(char-code <chr>)	GET THE ASCII CODE OF A CHARACTER
<chr>	the character
returns	the ASCII character code (integer, *parity bit stripped*)

(code-char <code>)	GET THE CHARACTER WITH A SPECFIED ASCII CODE
<code>	the ASCII code (integer, *range 0-127*)
returns	the character with that code or nil

(char-upcase <chr>)	CONVERT A CHARACTER TO UPPER CASE
<chr>	the character
returns	the upper case character

(char-downcase <chr>)	CONVERT A CHARACTER TO LOWER CASE
<chr>	the character
returns	the lower case character

(digit-char <n>)	CONVERT A DIGIT WEIGHT TO A DIGIT
<n>	the digit weight (integer)
returns	the digit character or nil

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

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

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

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

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

<chr1> the first character to compare
<chr2> the second character(s) to compare
returns t if predicate is true, nil otherwise
Note: case is significant with these comparison functions.

(char-lessp <chr1> <chr2>...)
(char-not-greaterp <chr1> <chr2>...)
(char-equalp <chr1> <chr2>...)
(char-not-equalp <chr1> <chr2>...)
(char-not-lessp <chr1> <chr2>...)
(char-greaterp <chr1> <chr2>...)

<chr1> the first string to compare
<chr2> the second string(s) to compare
returns t if predicate is true, nil otherwise
Note: case is not significant with these comparison functions.

STRUCTURE FUNCTIONS

(Added feature from XLISP 2.1)

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

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

or

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

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

The recognized options are:

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

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

Each slot description takes the form:

<name>

or

(<name> <defexpr>)

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

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

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

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

For instance:

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

creates the following functions:

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

OBJECT FUNCTIONS

This section is completely new. Note that the functions provided in classes.lsp are useful but not necessary.

Messages defined for Object and Class are listed starting on page 16.

(send <object> <message> [<args>...]) SEND A MESSAGE

<object>	the object to receive the message
<message>	message sent to object
<args>	arguments to method (if any)
returns	the result of the method

(send-super <message> [<args>]) SEND A MESSAGE TO SUPERCLASS

valid only in method context

<message>	message sent to objects superclass
<args>	arguments to method (if any)
returns	the result of the method

(defclass <sym> <ivars> [<cvars> [<super>]]) DEFINE A NEW CLASS

defined in class.lsp as a macro

<sym>	symbol whose value is to be bound to the class object (quoted)
<ivars>	list of instance variables (quoted). Instance variables specified either as <ivar> or (<ivar> <init>) to specify non-nil default initial value.
<cvars>	list of class variables (quoted)
<super>	superclass, or Object if absent.

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

Methods defined for classes defined with defclass:

(send <object> :<ivar>)

Returns the specified instance variable

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

Used to set an instance variable, typically with setf.

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

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

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

defined in class.lsp as a macro

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

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

defined in class.lsp as a macro

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

PREDICATE FUNCTIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(evenp <expr>) IS THIS INTEGER EVEN?
<expr> the integer to test
returns t if the integer is even, nil otherwise

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

(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?

Note: works with all numbers

<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

fsubr

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

returns the value of the first expression whose predicate is not nil

(and <expr>...) THE LOGICAL AND OF A LIST OF EXPRESSIONS

fsubr

<expr> the expressions to be ANDed

returns nil if any expression evaluates to nil, otherwise the value of the last expression (evaluation of expressions stops after the first expression that evaluates to nil)

(or <expr>...) THE LOGICAL OR OF A LIST OF EXPRESSIONS

fsubr

<expr> the expressions to be ORed

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

(if <texpr> <expr1> [<expr2>]) EVALUATE EXPRESSIONS CONDITIONALLY

fsubr

<texpr> the test expression

<expr1> the expression to be evaluated if texpr is non-nil

<expr2> the expression to be evaluated if texpr is nil

returns the value of the selected expression

(when <texpr> <expr>...) EVALUATE ONLY WHEN A CONDITION IS TRUE

fsubr

<texpr> the test expression

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

returns the value of the last expression or nil

(unless <texpr> <expr>...) EVALUATE ONLY WHEN A CONDITION IS FALSE

fsubr

<texpr> the test expression

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

returns the value of the last expression or nil (case <expr> <case>...[(t <expr>)]) SELECT BY CASE

fsubr

<expr> the selection expression

<case> pair consisting of:
(<value> <expr>...)
where:
<value> is a single expression or a list of expressions (unevaluated)
<expr> are expressions to execute if the case matches

(t <expr>) default case (no previous matching)

returns the value of the last expression of the matching case

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

fsubr

<binding> the variable bindings each of which is either:
1) a symbol (which is initialized to nil)
2) a list whose car is a symbol and whose cadr is an initialization expression

<expr> the expressions to be evaluated

returns the value of the last expression

(flet (<binding>...) <expr>...) CREATE LOCAL FUNCTIONS
(labels (<binding>...) <expr>...) FLET WITH RECURSIVE FUNCTIONS
(macrolet (<binding>...) <expr>...) CREATE LOCAL MACROS

fsubr

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

<expr> the expressions to be evaluated

returns the value of the last expression

(catch <sym> <expr>...) EVALUATE EXPRESSIONS AND CATCH THROWS

fsubr

<sym> the catch tag

<expr> expressions to evaluate

returns the value of the last expression the throw expression

(throw <sym> [<expr>]) THROW TO A CATCH

fsubr

<sym> the catch tag

<expr> the value for the catch to return (defaults to nil)

returns never returns

(unwind-protect <expr> <cexpr>...) PROTECT EVALUATION OF AN EXPRESSION

fsubr

<expr>	the expression to protect
<cexpr>	the cleanup expressions
returns	the value of the expression

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

LOOPING CONSTRUCTS

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

fsubr

<expr>	the body of the loop
returns	never returns (must use non-local exit)

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

fsubr. do binds simultaneously, do binds sequentially*

<binding>	the variable bindings each of which is either:		
	1)	a symbol (which is initialized to nil)	
	2)	a list of the form: (<sym> <init> [<step>])	
	where:		
		<sym>	is the symbol to bind
		<init>	the initial value of the symbol
		<step>	a step expression
<texpr>	the termination test expression		
<rexpr>	result expressions (the default is nil)		
<expr>	the body of the loop (treated like an implicit prog)		
returns	the value of the last result expression		

(dolist (<sym> <expr> [<rexpr>]) <expr>...) LOOP THROUGH A LIST

fsubr

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

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

fsubr

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

THE PROGRAM FEATURE

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

fsubr

<binding>	the variable bindings each of which is either:
	1) a symbol (which is initialized to nil)
	2) a list whose car is a symbol and whose cadr is an initialization expression
<expr>	expressions to evaluate or tags (symbols)
returns	nil or the argument passed to the return function

(block <name> <expr>...) NAMED BLOCK

fsubr

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

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

fsubr

<expr>	the value (defaults to nil)
returns	never returns

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

fsubr

<name>	the block *or function* name (quoted symbol)
<value>	the value to return (defaults to nil)
returns	never returns

(tagbody <expr>...) BLOCK WITH LABELS

fsubr

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

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

fsubr

<sym>	the tag (quoted)
returns	never returns

(progv <slist> <vlist> <expr>...) DYNAMICALLY BIND SYMBOLS

fsubr

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

(prog1 <expr1> <expr>...) EXECUTE EXPRESSIONS SEQUENTIALLY

fsubr

<expr1>	the first expression to evaluate
<expr>	the remaining expressions to evaluate
returns	the value of the first expression

(prog2 <expr1> <expr2> <expr>...) EXECUTE EXPRESSIONS SEQUENTIALLY

fsubr

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

(progn <expr>...) EXECUTE EXPRESSIONS SEQUENTIALLY

fsubr

<expr>	the expressions to evaluate
returns	the value of the last expression (or nil)

INPUT/OUTPUT FUNCTIONS

Note that when printing objects, printing is accomplished by sending the message :prin1 to the object (this is a modification of the distribution).

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

<stream>	the input stream (default is *standard-input*)
<eof>	the value to return on end of file (default is nil)
<rflag>	recursive read flag (default is nil, *unused in this implementation*)
returns	the expression read

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

defined in init.lsp

<ch>	*character to define*
<fcn>	*function to bind to character (see page 11)*
T	*if TMACRO rather than NMACRO*

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

defined in init.lsp

<ch>	*character*
returns	*function bound to character*

(print <expr> [<stream>]) PRINT AN EXPRESSION ON A NEW LINE

<expr>	the expression to be printed
<stream>	the output stream (default is *standard-output*)
returns	the expression

(prin1 <expr> [<stream>]) PRINT AN EXPRESSION

symbols, cons cells (without circularities), arrays, strings, and characters are printed in a format acceptable to the read function.

<expr>	the expression to be printed
<stream>	the output stream (default is *standard-output*)
returns	the expression

(princ <expr> [<stream>]) PRINT AN EXPRESSION WITHOUT QUOTING

<expr>	the expressions to be printed
<stream>	the output stream (default is *standard-output*)
returns	the expression

(pprint <expr> [<stream>]) PRETTY PRINT AN EXPRESSION

<expr>	the expressions to be printed
<stream>	the output stream (default is *standard-output*)
returns	the expression

(terpri [<stream>]) TERMINATE THE CURRENT PRINT LINE

<stream> the output stream (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

THE FORMAT FUNCTION

(format <stream> <fmt> <arg>...) DO FORMATTED OUTPUT

<stream>	the output stream
<fmt>	the format string
<arg>	the format arguments
returns	output string if <stream> is nil, nil otherwise

*If <stream> is T, then *STANDARD-OUTPUT* is assumed.*

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

~A *or ~a*	print next argument using princ
~S *or ~s*	print next argument using prin1
~%	start a new line
~~	print a tilde character
~\n	*ignore return and following whitespace*

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

For ~A and ~S the full form is:

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

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

~15,,2,'.@A

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

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

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

FILE I/O FUNCTIONS

*Note that initially, when starting xlisp, there are five system stream symbols which are associated with three streams. *STANDARD-INPUT* is bound to standard input and *STANDARD-OUTPUT* is bound to standard output. *ERROR-OUTPUT* (error message output), *TRACE-OUTPUT* (from TRACE function), and *DEBUG-IO* (break loop i/o) are all bound to standard error, which is just about always the console.*

When the transcript is active (either -t on the command line or the DRIBBLE function), all characters that would be sent to the display via either standard output or standard error are also placed in the transcript file.

XLISP gives special treatment to i/o through standard input/output/error, as shown on page 3. An XLISP program which alters any of the system stream values should save the original values to restore them. Using the OPEN function to re-open the console will not restore i/o through standard input/output/error.

(open <fname> &key :direction *:element-type*) OPEN A FILE STREAM

<fname> the file name string or symbol

:direction :input, :output, or *:io* (default is :input). A file opened for input must already exist. A file opened for output will delete any existing file by the same name. A file opened for input or io is positioned at its start.

:element-type FIXNUM or CHARACTER (default is CHARACTER), as returned by type-of function. Files opened with type FIXNUM are binary files instead of ascii, which means no crlf to/from lf conversion takes place, and control-Z will not terminate an input file. It is the intent of Common Lisp that binary files only be accessed with read-byte and write-byte while ascii files be accessed with any function but read-byte and write-byte. XLISP does not enforce that distinction.

returns a file stream

(close <stream>) CLOSE A FILE STREAM

<stream> the file stream

returns nil

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

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

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

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

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

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

<stream>	*the file stream (should be a disk file)*
<expr>	*desired file position, if setting position*
returns	*if setting position, and successful, then T; if getting position and successful then the position; otherwise NIL*

(read-char [<stream>]) READ A CHARACTER FROM A STREAM

<stream>	the input stream (default is *standard-input*)
returns	the character

(peek-char [<flag> [<stream>]]) PEEK AT THE NEXT CHARACTER

<flag>	flag for skipping white space (default is nil)
<stream>	the input stream (default is *standard-input*)
returns	the character (integer)

(write-char <ch> [<stream>]) WRITE A CHARACTER TO A STREAM

<ch>	the character to write
<stream>	the output stream (default is *standard-output*)
returns	the character

(read-line [<stream>]) READ A LINE FROM A STREAM

<stream>	the input stream (default is *standard-input*)
returns	the string

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

<stream>	the input stream (default is *standard-input*)
returns	the byte (integer)

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

<byte>	the byte to write (integer)
<stream>	the output stream (default is *standard-output*)
returns	the byte (integer)

STRING STREAM FUNCTIONS

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

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

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

(make-string-input-stream <str> [<start> [<end>]])
- <str> the string
- <start> the starting offset
- <end> the ending offset + 1 *or NIL for end of string*
- returns an unnamed stream that reads from the string

(make-string-output-stream)
- returns an unnamed output stream

(get-output-stream-string <stream>)
- <stream> the output stream
- returns the output so far as a string
- Note: the output stream is emptied by this function

(get-output-stream-list <stream>)
- <stream> the output stream
- returns the output so far as a list
- Note: the output stream is emptied by this function

DEBUGGING AND ERROR HANDLING FUNCTIONS

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

fsubr

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

(untrace [<sym>...]) REMOVE A FUNCTION FROM THE TRACE LIST

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

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

(error <emsg> [<arg>]) SIGNAL A NON-CORRECTABLE ERROR

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

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

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

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

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

(clean-up) CLEAN-UP AFTER AN ERROR

returns	never returns

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

returns	never returns

(continue) CONTINUE FROM A CORRECTABLE ERROR

returns	never returns

(errset <expr> [<pflag>]) TRAP ERRORS

fsubr

<expr>	the expression to execute
<pflag>	flag to control printing of the error message
returns	the value of the last expression consed with nil or nil on error

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

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

(evalhook <expr> <ehook> <ahook> [<env>]) EVALUATE WITH HOOKS

<expr> the expression to evaluate. *<ehook> is not used at the top level.*
<ehook> the value for *evalhook*
<ahook> the value for *applyhook*
<env> the environment (default is nil). *The format is a dotted pair of value (car) and function (cdr) binding lists. Each binding list is a list of level binding a-lists, with the innermost a-list first. The level binding a-list associates the bound symbol with its value.*
returns the result of evaluating the expression

(applyhook <fun> <arglist> <ehook> <ahook>) APPLY WITH HOOKS

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

(debug) ENABLE DEBUG BREAKS
(nodebug) DISABLE DEBUG BREAKS

Defined in init.lsp

SYSTEM FUNCTIONS

(load <fname> &key :verbose :print) LOAD A SOURCE FILE
- <fname> the filename string or symbol
- :verbose the verbose flag (default is t)
- :print the print flag (default is nil)
- returns the filename

(restore <fname>) RESTORE WORKSPACE FROM A FILE
- <fname> the filename string or symbol
- returns nil on failure, otherwise never returns

(save <fname>) SAVE WORKSPACE TO A FILE
- <fname> the filename string or symbol
- returns t if workspace was written, nil otherwise

(savefun <fcn>) SAVE FUNCTION TO A FILE
- *defined in init.lsp*
- *<fcn> function name (saves it to file of same name, with extension ".lsp")*
- *returns t if successful*

(dribble [<fname>]) CREATE A FILE WITH A TRANSCRIPT OF A SESSION
- <fname> file name string 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, default 1
- 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

(room) SHOW MEMORY ALLOCATION STATISTICS
- returns nil

(time <expr>) MEASURE EXECUTION TIME
- *fsubr. Note: added function*
- *<expr> the expression to evaluate*
- *returns the execution time, in seconds (floating point)*

(coerce <expr> <type>) *FORCE EXPRESSION TO DESIGNATED TYPE*

Note: added function

<expr>	*the expression to coerce*
<type>	*desired type, as returned by type-of*
returns	*<expr> if type is correct, or converted object.*

Sequences can be coerced into other sequences, single character strings or symbols with single character printnames can be coerced into characters, integers can be coerced into characters or flonums.

(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-in functions
	FSUBR	for special forms
	CLOSURE	for defined functions
	STRING	for strings
	FIXNUM	for integers
	FLONUM	for floating point numbers
	CHARACTER	for characters
	FILE-STREAM	for file pointers
	UNNAMED-STREAM	for unnamed streams
	ARRAY	for arrays
	sym	*for structures of type "sym"*

(generic <expr>) *CREATE A GENERIC TYPED COPY OF THE EXPRESSION*

Note: added function, Tom Almy's invention.

<expr>	*the expression to copy*	
returns	*nil if value is nil, otherwise if type is:*	
	SYMBOL	*copy as an ARRAY*
	OBJECT	*copy as an ARRAY*
	CONS	*(CONS (CAR <expr>)(CDR <expr>))*
	CLOSURE	*copy as an ARRAY*
	STRING	*copy of the string*
	FIXNUM	*value*
	FLONUM	*value*
	CHARACTER	*value*
	UNNAMED-STREAM	*copy as a CONS*
	ARRAY	*copy of the array*

(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)

(getkey) READ A KEYSTROKE FROM CONSOLE

returns	*integer value of key (no echo)*

(system <command>) EXECUTE A SYSTEM COMMAND

<command>	*Command string, if 0 length then spawn OS shell*
returns	*T if successful (note that MS/DOS command.com always returns success)*

(exit) EXIT XLISP

returns	*never returns*

The following graphic functions represent an extension by Tom Almy:

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

<ax>	Graphic mode (value passed in register AX)
<bx>	BX value for some extended graphic modes
<width>	width for extended graphic modes
<height>	height for extended graphic modes
returns	T

(color <value>) SET DRAWING COLOR

<value>	Drawing color (not checked for validity)
returns	<value>

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

For moverel, all coordinates are relative to the preceeding point.

<x1> <y1>	Moves to point x1,y1 in anticipation of draw.
<x2> <y2>	Draws to points specified in additional arguments.
returns	T if succeeds, else NIL

(draw [<x1> <y1> ...]) ABSOLUTE DRAW
(drawrel [<x1> <y1> ...]) RELATIVE DRAW

For drawrel, all coordinates are relative to the preceeding point.

<x1> <y1>	Point(s) drawn to, in order.
returns	T if succeeds, else NIL

ADDITIONAL FUNCTIONS AND UTILITIES

This section is completely new (added by Tom Almy).

STEP.LSP

This file contains a simple Lisp single-step debugger. It started as an implementation of the "hook" example in chapter 20 of Steele's "Common Lisp". This version was brought up on Xlisp 1.7 for the Amiga, and then on VAXLISP.

To invoke: (step (whatever-form with args))

For each list (interpreted function call), the stepper prints the environment and the list, then enters a read-eval-print loop. At this point the available commands are:

(a list)<CR>	evaluate the list in the current environment, print the result, and repeat.
<CR>	step into the called function
anything_else<CR>	step over the called function.

If the stepper comes to a form that is not a list it prints the form and the value, and continues on without stopping.

Note that stepper commands are executed in the current environment. Since this is the case, the stepper commands can change the current environment. For example, a SETF will change an environment variable and thus can alter the course of execution.

Global variables - newline, *hooklevel*

Functions/macros - while step eval-hool-function step-spaces step-flush

Note — an even more powerful stepper package is in stepper.lsp (documented in stepper.doc).

PP.LSP

In addition to the pretty-printer itself, this file contains a few functions that illustrate some simple but useful applications.

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

<object>	The expression to print
<funct>	Function to print (as DEFUN or DEFMACRO)
<file>	File to print (specify either as string or quoted symbol)
<stream>	Output stream (default is *standard-output*)
returns	T

Global variables: tabsize maxsize miser-size min-miser-car max-normal-car

Functions/Macros: sym-function pp-file pp-def make-def pp pp1 moveto spaces pp-rest-across pp-rest printmacrop pp-binding-form pp-do-form pp-defining-form pp-pair-form

See the source file for more information.

REPAIR.LSP

This file contains a structure editor.
Execute (repair 'symbol) to edit a symbol.
or (repairf symbol) to edit the function binding of a symbol (allows changing the argument list or function type, lambda or macro).

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

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

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

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

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

(command list on next page)

COMMANDS (general):

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

COMMANDS (if selection is symbol):

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

COMMANDS (if selection is list):

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

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

EXAMPLES: FILE I/O FUNCTIONS

Input from a File

To open a file for input, use the OPEN function with the keyword argument :DIRECTION set to :INPUT. To open a file for output, use the OPEN function with the keyword argument :DIRECTION set to :OUTPUT. The OPEN function takes a single required argument which is the name of the file to be opened. This name can be in the form of a string or a symbol. The OPEN function returns an object of type FILE-STREAM if it succeeds in opening the specified file. It returns the value NIL if it fails. In order to manipulate the file, it is necessary to save the value returned by the OPEN function. This is usually done by assigning it to a variable with the SETQ special form or by binding it using LET or LET*. Here is an example:

```
(setq fp (open "init.lsp" :direction :input))
```

Evaluating this expression will result in the file "init.lsp" being opened. The file object that will be returned by the OPEN function will be assigned to the variable "fp".

It is now possible to use the file for input. To read an expression from the file, just supply the value of the "fp" variable as the optional "stream" argument to READ.

```
(read fp)
```

Evaluating this expression will result in reading the first expression from the file "init.lsp". The expression will be returned as the result of the READ function. More expressions can be read from the file using further calls to the READ function. When there are no more expressions to read, the READ function will return NIL (or whatever value was supplied as the second argument to READ).

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

```
(close fp)
```

Evaluating this expression will cause the file to be closed.

Output to a File

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

```
(setq fp (open "test.dat" :direction :output))
```

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

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

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

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

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

```
(close fp)
```

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

A Slightly More Complicated File Example

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

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

INDEX