XLISP–PLUS

Reference Manual

Version 3.0

XLISP-PLUS: Another Object-oriented Lisp

Version 3.0

February 15, 2016

Tom Almy
tom@almy.us

TABLE OF CONTENTS

INTRODUCTION

XLISP-PLUS is an enhanced version of David Michael Betz's XLISP to have additional features of Common Lisp. XLISP-PLUS is distributed for Microsoft Windows, Apple OS X, Linux and UNIX, but can be easily ported to other platforms. Complete source code is provided (in "C") to allow easy modification and extension.

Since XLISP-PLUS is based on XLISP, most XLISP programs will run on XLISP-PLUS. Since XLISP-PLUS incorporates many more features of Common Lisp, many small Common Lisp applications will run on XLISP-PLUS with little modification. See the section starting on page 152 for details of the differences between XLISP and XLISP-PLUS and the index entries of Compatibility with previous versions.

Many Common Lisp functions are built into XLISP-PLUS. In addition, XLISP defines the objects *Object* and *Class* as primitives. *Object* is the only class that has no superclass and hence is the root of the class hierarchy tree. *Class* is the class of which all classes are instances (it is the only object that is an instance of itself).

This document is a brief description of XLISP-PLUS. It assumes some knowledge of Lisp and some understanding of the concepts of object-oriented programming.

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

XLISP-PLUS has a number of compilation options to eliminate groups of functions and to tailor itself to various environments. Unless otherwise indicated this manual assumes all options are enabled and the system dependent code is as complete as that provided for the MS-DOS environment. Assistance for using or porting XLISP-PLUS can be obtained by writing to Tom Almy at the address tom@almy.us, website http://almy.us/xlisp.html.

GETTING STARTED

The XLISP-PLUS archives are available via the XLISP-PLUS Web Page at: http://almy.us/xlisp.html.

The archives are the following:

Archive XL304SRC: Contains all source files. Consult the README.SRC file in that archive for additional information.

Archive XL304DOC: Contains ASCII documentation for XLISP-PLUS. *Not current.*

Archive XL304PSD: Contains the present documentation in Acrobat format (using the fonts Times Roman, Helvetica and Courier).

Archive XL300REQ: Contains this file, update file (README.UPD), information about the Microsoft Windows version (README.WIN), tutorial (LISP-TUT.DOC) and all the Lisp source files REQuired for operation (see README.LSP).

Archive XL300EXE: Executable program which runs on any generic IBM/PC or Clone. Compiled using Borland C++ 4.0 (as a C compiler), Ralf Brown's spawn version 3.0 for nice SYSTEM function, and Fabrice Bellard's lzexe to compact the executable. It's tight for memory, so any of the versions that follow are preferable instead.

Archive XL300386: Executable program which requires 80386 or 80486, and at least 1 megabyte of extended memory. Uses DJ Delorie's port of the GNU C compiler, and his DOS extender, GO32. Compatible with DPMI, VCPI, and XMS. This archive also contains EMU387, the emulator for above. To use, set environment variable "set go32=emu c:/bin/emu387" assuming this file is in c:\bin directory. YOU MUST USE THIS EMULATOR FOR XLISP.EXE ON SYSTEMS WITHOUT THE FLOATING POINT COPROCESSOR (486SX, 386 without 387, and a number of non-Intel 486 variations).

Archive XL300OS2: Executable program for OS/2 V2.x or later. Uses GCC and EMX.

Archive XL300WIN: Executable for Microsoft Windows 3.1, using an 80386 or better. Also WIN32S 32 bit executable for Windows 3.1, OS/2 WARP, Windows/95, and Windows/NT. See file readme.win for details.

For MS-DOS:

Note that old versions (such as 80286, or 80x87 support for real mode) are not supported any more.

The archives have to be decompressed using a standard decompressing program, such as PKUNZIP. In the following examples, we will assume that all the archives are stored in the subdirectory C:\XLISP.

You should issue the following commands:

```
C:
CD \XLISP
PKUNZIP *.ZIP
```

The following will set the OS variable XLPATH, which indicates the subdirectories in which the XLISP interpreter should search when loading Lisp or system files:

```
SET XLPATH=.;C:\XLISP\;
```

Adding the above line to the AUTOEXEC.BAT file, will save you typing it every time you want to run XLISP.

Finally run the interpreter:

XLISP

When you want to exit the XLISP interpreter and go back to the OS, type the following Lisp command:

(exit)

Other platforms:

Mac OS X

Unix/Linux

AMIGA: Now available (but untested).

IBM Mainframe (370): Now available (but untested).

Other computers: The code was made as portable as possible. You'll need to write a *STUFF.C file for your system. Also read XLISP.H closely. All the 80x86'isms are conditionally included, so these won't get in the way.

C++: It won't compile, but you should be able to hack on the code a while and get it to go. The ANSI C conversion carried out should help you, although function headers have been converted only as necessary to keep the C compilers happy.

A QUICK LISP TUTORIAL

Here is a short tutorial of Common Lisp to get you started. If you are an experienced Lisp programmer, you can skip to the next section. For a complete reference of the language you will need the book "Common Lisp: the language", by G. L. Steele, Jr.

These are the main elements of Lisp:

Symbols

A symbol is just a string of characters. There are restrictions on what you can include in a symbol and what the first character can be, but as long as you stick to letters, digits, and hyphens, you'll be safe.

(Except that if you use only digits and possibly an initial hyphen, Lisp will think you typed an integer rather than a symbol.) Some examples of symbols:

```
  a
  b
  c1
  foo
  bar
  baaz-quux-garply
```

Some things you can do with symbols follow. (Things after a ">" prompt are what you type to the Lisp interpreter, while other things are what the Lisp interpreter prints back to you. The ";" is Lisp's comment character: everything from a ";" to the end of line is ignored.)

```
> (setq a 5)            ;store a number as the value of a symbol
5
> a                     ;take the value of a symbol
5
> (let ((a 6)) a)       ;bind the value of a symbol temporarily to 6
6
> a                     ;the value returns to 5 once the let is finished
5
> (+ a 6)               ;use the value of a symbol as an argument to a function
11
> b                     ;try to take the value of a symbol which has no value
error: unbound variable - b
if continued: try evaluating symbol again
```

There are two special symbols, t and nil. The value of t is defined always to be t, and the value of nil is defined always to be nil. Lisp uses t and nil to represent true and false. An example of this use is in the if statement, described more fully later:

```
> (if t 5 6)
5
> (if nil 5 6)
6
> (if 4 5 6)
5
```

The last example is odd but correct: nil means false, and anything else means true. (Unless we have a reason to do otherwise, we use t to mean true, just for the sake of clarity.)

Symbols like t and nil are called self-evaluating symbols, because they evaluate to themselves. There is a whole class of self-evaluating symbols called keywords; any symbol whose name starts with a colon is a keyword. (See below for some uses for keywords.) Some examples:

```
> :this-is-a-keyword
:THIS-IS-A-KEYWORD
> :so-is-this
:SO-IS-THIS
> :me-too
:ME-TOO
```

Numbers

An integer is a string of digits optionally preceded by + or -. A real number looks like an integer, except that it has a decimal point and optionally can be written in scientific notation. A rational looks like two integers with a / between them. Lisp supports complex numbers, which are written #c(r i) (where r is the real part and i is the imaginary part). A number is any of the above. Here are some numbers:

```
  5
  17
  -34
  +6
  3.1415
  1.722e-15
  #c(1.722e-15 0.75)
```

The standard arithmetic functions are all available: +, -, *, /, floor, ceiling, mod, sin, cos, tan, sqrt, exp, expt, and so forth. All of them accept any kind of number as an argument. +, -, *, and / return a number according to type contagion: an integer plus a rational is a rational, a rational plus a real is a real, and a real plus a complex is a complex. Here are some examples:

```
> (+ 3 3/4)             ;type contagion
15/4
> (exp 1)               ;e
2.71828
> (exp 3)               ;e*e*e
20.0855
> (expt 3 4.2)          ;exponent with a base other than e
100.904
> (+ 5 6 7 (* 8 9 10))  ;the fns +-*/ all accept multiple arguments
```

There is no limit to the absolute value of an integer except the memory size of your computer. Be warned that computations with bignums (as large integers are called) can be slow. (So can computations with rationals, especially compared to the corresponding computations with small integers or floats.)

Conses

A cons is just a two-field record. The fields are called "car" and "cdr", for historical reasons. (On the first machine where Lisp was implemented, there were two instructions CAR and CDR which stood for "contents of address register" and "contents of decrement register". Conses were implemented using these two registers.)

Conses are easy to use:

```
> (cons 4 5)            ;Allocate a cons. Set the car to 4 and the cdr to 5.
(4 . 5)
> (cons (cons 4 5) 6)
((4 . 5) . 6)
> (car (cons 4 5))
4
> (cdr (cons 4 5))
5
```

Lists

You can build many structures out of conses. Perhaps the simplest is a linked list: the car of each cons points to one of the elements of the list, and the cdr points either to another cons or to nil. You can create such a linked list with the list function:

```
> (list 4 5 6)
(4 5 6)
```

Notice that Lisp prints linked lists in a special way: it omits some of the periods and parentheses. The rule is: if the cdr of a cons is nil, Lisp doesn't bother to print the period or the nil; and if the cdr of cons A is cons B, then Lisp doesn't bother to print the period for cons A or the parentheses for cons B. So:

```
> (cons 4 nil)
(4)
> (cons 4 (cons 5 6))
(4 5 . 6)
> (cons 4 (cons 5 (cons 6 nil)))
(4 5 6)
```

The last example is exactly equivalent to the call (list 4 5 6). Note that nil now means the list with no elements: the cdr of (a b), a list with 2 elements, is (b), a list with 1 element; and the cdr of (b), a list with 1 element, is nil, which therefore must be a list with no elements.

The car and cdr of nil are defined to be nil.

If you store your list in a variable, you can make it act like a stack:

```
> (setq a nil)
NIL
> (push 4 a)
(4)
> (push 5 a)
(5 4)
> (pop a)
5
> a
(4)
> (pop a)
4
> (pop a)
NIL
> a
NIL
```

Functions

You saw some examples of functions above. Here are some more:

```
> (+ 3 4 5 6)                   ;this function takes any number of arguments
18
> (+ (+ 3 4) (+ (+ 4 5) 6))     ;isn't prefix notation fun?
22
> (defun foo (x y) (+ x y 5))   ;defining a function
FOO
> (foo 5 0)                     ;calling a function
10
> (defun fact (x)               ;a recursive function
    (if (> x 0)
      (* x (fact (- x 1)))
      1
  ) )
FACT
> (fact 5)
120
> (defun a (x) (if (= x 0) t (b (- x))))        ;mutually recursive functions
A
> (defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
B
> (a 5)
T
> (defun bar (x)                ;a function with multiple statements in
    (setq x (* x 3))            ;its body -- it will return the value
    (setq x (/ x 2))            ;returned by its final statement
    (+ x 4)
  )
BAR
> (bar 6)
13
```

When we defined foo, we gave it two arguments, x and y. Now when we call foo, we are required to provide exactly two arguments: the first will become the value of x for the duration of the call to foo, and the second will become the value of y for the duration of the call. In Lisp, most variables are lexically scoped; that is, if foo calls bar and bar tries to reference x, bar will not get foo's value for x.

The process of assigning a symbol a value for the duration of some lexical scope is called binding.

You can specify optional arguments for your functions. Any argument after the symbol &optional is optional:

```
> (defun bar (x &optional y) (if y x 0))
BAR
> (defun baaz (&optional (x 3) (z 10)) (+ x z))
BAAZ
> (bar 5)
0
> (bar 5 t)
5
> (baaz 5)
15
> (baaz 5 6)
11
> (baaz)
13
```

It is legal to call the function bar with either one or two arguments. If it is called with one argument, x will be bound to the value of that argument and y will be bound to nil; if it is called with two arguments, x and y will be bound to the values of the first and second argument, respectively.

The function baaz has two optional arguments. It specifies a default value for each of them: if the caller specifies only one argument, z will be bound to 10 instead of to nil, and if the caller specifies no arguments, x will be bound to 3 and z to 10.

You can make your function accept any number of arguments by ending its argument list with an &rest parameter. Lisp will collect all arguments not otherwise accounted for into a list and bind the &rest parameter to that list. So:

```
> (defun foo (x &rest y) y)
FOO
> (foo 3)
NIL
> (foo 4 5 6)
(5 6)
```

Finally, you can give your function another kind of optional argument called a keyword argument. The caller can give these arguments in any order, because they're labeled with keywords.

```
> (defun foo (&key x y) (cons x y))
FOO
> (foo :x 5 :y 3)
(5 . 3)
> (foo :y 3 :x 5)
(5 . 3)
> (foo :y 3)
(NIL . 3)
> (foo)
(NIL)
```

An &key parameter can have a default value too:

```
> (defun foo (&key (x 5)) x)
FOO
> (foo :x 7)
7
> (foo)
5
```

Printing

Some functions can cause output. The simplest one is print, which prints its argument and then returns it.

```
> (print 3)
3
3
```

The first 3 above was printed, the second was returned.

If you want more complicated output, you will need to use format.

Here's an example:

```
> (format t "An atom: ~S~%and a list: ~S~%and an integer: ~D~%"
          nil (list 5) 6)
An atom: NIL
and a list: (5)
and an integer: 6
```

The first argument to format is either t, nil, or a stream. T specifies output to the terminal. Nil means not to print anything but to return a string containing the output instead. Streams are general places for output to go: they can specify a file, or the terminal, or another program. This handout will not describe streams in any further detail.

The second argument is a formatting template, which is a string optionally containing formatting directives.

All remaining arguments may be referred to by the formatting directives. Lisp will replace the directives with some appropriate characters based on the arguments to which they refer and then print the resulting string.

Format always returns nil unless its first argument is nil, in which case it prints nothing and returns a string.

There are three different directives in the above example: ~S, ~D, and ~%. The first one accepts any Lisp object and is replaced by a printed representation of that object (the same representation which is produced by print). The second one accepts only integers. The third one doesn't refer to an argument; it is always replaced by a carriage return.

Another useful directive is ~~, which is replaced by a single ~.

Refer to page 99 of the manual for many additional formatting directives.

Forms and the Top-Level Loop

The things which you type to the Lisp interpreter are called forms; the Lisp interpreter repeatedly reads a form, evaluates it, and prints the result. This procedure is called the read-eval-print loop.

Some forms will cause errors. After an error, Lisp will put you into the debugger so you can try to figure out what caused the error. Lisp debuggers are all different; but most will respond to the command "help" or ":help" by giving some form of help. In the XLISP-PLUS package there are provided two debuggers called step.lsp and stepper.lsp (see pages 117 and 118).

In general, a form is either an atom (for example, a symbol, an integer, or a string) or a list. If the form is an atom, Lisp evaluates it immediately. Symbols evaluate to their value; integers and strings evaluate to themselves. If the form is a list, Lisp treats its first element as the name of a function; it evaluates the remaining elements recursively, and then calls the function with the values of the remaining elements as arguments.

For example, if Lisp sees the form (+ 3 4), it treats + as the name of a function. It then evaluates 3 to get 3 and 4 to get 4; finally it calls + with 3 and 4 as the arguments. The + function returns 7, which Lisp prints.

The top-level loop provides some other conveniences; one particularly convenient convenience is the ability to talk about the results of previously typed forms. Lisp always saves its most recent three results; it stores them as the values of the symbols *, **, and ***. For example:

```
> 3
3
> 4
4
> 5
5
> ***
3
> ***
4
> ***
5
> **
4
> *
4
```

Special forms

There are a number of special forms which look like function calls but aren't. These include control constructs such as if statements and do loops; assignments like setq, setf, push, and pop; definitions such as defun and defstruct; and binding constructs such as let. (Not all of these special forms have been mentioned yet. See below.)

One useful special form is the quote form: quote prevents its argument from being evaluated. For example:

```
> (setq a 3)
3
> a
3
> (quote a)
A
> 'a                    ;'a is an abbreviation for (quote a)
A
```

Another similar special form is the function form: function causes its argument to be interpreted as a function rather than being evaluated.

For example:

```
> (setq + 3)
3
> +
3
> '+
+
> (function +)
#<Subr-+: #88b44d5e>
> #'+                   ;#'+ is an abbreviation for (function +)
#<Subr-+: #88b44d5e>
```

The function special form is useful when you want to pass a function as an argument to another function. See below for some examples of functions which take functions as arguments.

Binding

Binding is lexically scoped assignment. It happens to the variables in a function's parameter list whenever the function is called: the formal parameters are bound to the actual parameters for the duration of the function call. You can bind variables anywhere in a program with the let special form, which looks like this:

```
(let ((var1 val1)
      (var2 val2)
      ...
     )
   body
   )
```

Let binds var1 to val1, var2 to val2, and so forth; then it executes the statements in its body. The body of a let follows exactly the same rules that a function body does. Some examples:

```
> (let ((a 3)) (+ a 1))
4
> (let ((a 2)
        (b 3)
        (c 0))
    (setq c (+ a b))
    c
  )
5
> (setq c 4)
4
> (let ((c 5)) c)
5
> c
4
```

Instead of (let ((a nil) (b nil)) …), you can write (let (a b) …).

The val1, val2, etc. inside a let cannot reference the variables var1, var2, etc… that the let is binding. For example,

```
> (let ((x 1)
        (y (+ x 1)))
    y
  )
error: unbound variable - x
```

If the symbol x already has a global value, stranger happenings will result:

```
> (setq x 7)
7
> (let ((x 1)
        (y (+ x 1)))
    y
  )
8
```

The let* special form is just like let except that it allows values to reference variables defined earlier in the let*. For example,

```
> (setq x 7)
7
> (let* ((x 1)
         (y (+ x 1)))
    y
  )
2
```

The form

```
(let* ((x a)
       (y b))
       ...
       )
```

is equivalent to

```
(let ((x a))
   (let ((y b))
         ...
         ) )
```

Dynamic Scoping

The let and let* forms provide lexical scoping, which is what you expect if you're used to programming in C or Pascal. Dynamic scoping is what you get in BASIC: if you assign a value to a dynamically scoped variable, every mention of that variable returns that value until you assign another value to the same variable.

In Lisp, dynamically scoped variables are called special variables. You can declare a special variable with the defvar special form. Here are some examples of lexically and dynamically scoped variables.

In this example, the function check-regular references a regular (ie, lexically scoped) variable. Since check-regular is lexically outside of the let which binds regular, check-regular returns the variable's global value.

```
> (setq regular 5)
5
> (defun check-regular () regular)
CHECK-REGULAR
> (check-regular)
5
> (let ((regular 6)) (check-regular))
5
```

In this example, the function check-special references a special (ie, dynamically scoped) variable. Since the call to check-special is temporally inside of the let which binds special, check-special returns the variable's local value.

```
> (defvar *special* 5)
*SPECIAL*
> (defun check-special () *special*)
CHECK-SPECIAL
> (check-special)
5
> (let ((*special* 6)) (check-special))
6
```

By convention, the name of a special variable begins and ends with a *. Special variables are chiefly used as global variables, since programmers usually expect lexical scoping for local variables and dynamic scoping for global variables.

For more information on the difference between lexical and dynamic scoping, see 'Common Lisp: the Language'.

Arrays

The function make-array makes a 1D array. The aref function accesses its elements. All elements of an array are initially set to nil. For example:

```
> (make-array 4)        ;1D arrays don't need the extra parens
#(NIL NIL NIL NIL)
```

Array indices always start at 0.

See below for how to set the elements of an array.

Strings

A string is a sequence of characters between double quotes. Lisp represents a string as a variable-length array of characters. You can write a string which contains a double quote by preceding the quote with a backslash; a double backslash stands for a single backslash. For example:

```
 "abcd" has 4 characters
 "\"" has 1 character
 "\\" has 1 character
```

Here are some functions for dealing with strings:

```
> (concatenate 'string "abcd" "efg")
"abcdefg"
> (char "abc" 1)
#\b                     ;Lisp writes characters preceded by #\
> (aref "abc" 1)
#\b                     ;remember, strings are really arrays
```

The concatenate function can actually work with any type of sequence:

```
> (concatenate 'string '(#\a #\b) '(#\c))
"abc"
> (concatenate 'list "abc" "de")
(#\a #\b #\c #\d #\e)
> (concatenate 'array '#(3 3 3) '#(3 3 3))
#(3 3 3 3 3 3)
```

Structures

Lisp structures are analogous to C structs or Pascal records. Here is an example:

```
> (defstruct foo
    bar
    baaz
    quux
  )
FOO
```

This example defines a data type called foo which is a structure containing 3 fields. It also defines 4 functions which operate on this data type: make-foo, foo-bar, foo-baaz, and foo-quux. The first one makes a new object of type foo; the others access the fields of an object of type foo. Here is how to use these functions:

```
> (make-foo)
#s(FOO BAR NIL BAAZ NIL QUUX NIL)
> (make-foo :baaz 3)
#s(FOO BAR NIL BAAZ 3 QUUX NIL)
> (foo-bar *)
NIL
> (foo-baaz **)
3
```

The make-foo function can take a keyword argument for each of the fields a structure of type foo can have. The field access functions each take one argument, a structure of type foo, and return the appropriate field.

See below for how to set the fields of a structure.

Setf

Certain forms in Lisp naturally define a memory location. For example, if the value of x is a structure of type foo, then (foo-bar x) defines the bar field of the value of x. Or, if the value of y is a one- dimensional array, (aref y 2) defines the third element of y.

The setf special form uses its first argument to define a place in memory, evaluates its second argument, and stores the resulting value in the resulting memory location. For example,

```
> (setq a (make-array 3))
#(NIL NIL NIL)
> (aref a 1)
NIL
> (setf (aref a 1) 3)
3
> a
#(NIL 3 NIL)
> (aref a 1)
3
> (defstruct foo bar)
FOO
> (setq a (make-foo))
#s(FOO :BAR NIL)
> (foo-bar a)
NIL
> (setf (foo-bar a) 3)
3
> a
#s(FOO :BAR 3)
> (foo-bar a)
3
```

Setf is the only way to set the fields of a structure or the elements of an array.

Here are some more examples of setf and related functions.

```
> (setf a (make-array 1))       ;setf on a variable is equivalent to setq
#(NIL)
> (push 5 (aref a 0))           ;push can act like setf
(5)
> (pop (aref a 0))              ;so can pop
5
> (setf (aref a 0) 5)
5
> (incf (aref a 0))             ;incf reads from a place, increments,
6                               ;and writes back
> (aref a 0)
6
```

Booleans and Conditionals

Lisp uses the self-evaluating symbol nil to mean false. Anything other than nil means true. Unless we have a reason not to, we usually use the self-evaluating symbol t to stand for true.

Lisp provides a standard set of logical functions, for example and, or, and not. The and and or connectives are short-circuiting: and will not evaluate any arguments to the right of the first one which evaluates to nil, while or will not evaluate any arguments to the right of the first one which evaluates to t.

Lisp also provides several special forms for conditional execution. The simplest of these is if. The first argument of if determines whether the second or third argument will be executed:

```
> (if t 5 6)
5
> (if nil 5 6)
6
> (if 4 5 6)
5
```

If you need to put more than one statement in the then or else clause of an if statement, you can use the progn special form. Progn executes each statement in its body, then returns the value of the final one.

```
> (setq a 7)
7
> (setq b 0)
0
> (setq c 5)
5
> (if (> a 5)
    (progn
      (setq a (+ b 7))
      (setq b (+ c 8)))
    (setq b 4)
  )
13
```

An if statement which lacks either a then or an else clause can be written using the when or unless special form:

```
> (when t 3)
3
> (when nil 3)
NIL
> (unless t 3)
NIL
> (unless nil 3)
3
```

When and unless, unlike if, allow any number of statements in their bodies (e.g., (when x a b c) is equivalent to (if x (progn a b c))).

```
> (when t
    (setq a 5)
    (+ a 6)
  )
11
```

More complicated conditionals can be defined using the cond special form, which is equivalent to an if … else if … fi construction.

A cond consists of the symbol cond followed by a number of cond clauses, each of which is a list. The first element of a cond clause is the condition; the remaining elements (if any) are the action. The cond form finds the first clause whose condition evaluates to true (ie, doesn't evaluate to nil); it then executes the corresponding action and returns the resulting value. None of the remaining conditions are evaluated; nor are any actions except the one corresponding to the selected condition. For example:

```
> (setq a 3)
3
> (cond
   ((evenp a) a)         ;if a is even return a
   ((> a 7) (/ a 2))     ;else if a is bigger than 7 return a/2
   ((< a 5) (- a 1))     ;else if a is smaller than 5 return a-1
   (t 17)                ;else return 17
  )
2
```

If the action in the selected cond clause is missing, cond returns what the condition evaluated to:

```
> (cond ((+ 3 4)))
7
```

Here's a clever little recursive function which uses cond. You might be interested in trying to prove that it terminates for all integers x greater than 1. (If you succeed, please publish the result.)

```
> (defun hotpo (x steps)        ;hotpo stands for Half Or Triple Plus One
    (cond
     ((= x 1) steps)
     ((oddp x) (hotpo (+ 1 (* x 3)) (+ 1 steps)))
     (t (hotpo (/ x 2) (+ 1 steps)))
  ) )
A
> (hotpo 7 0)
16
```

The Lisp case statement is like a C switch statement, or a Pascal case statement:

```
> (setq x 'b)
B
> (case x
   (a 5)
   ((d e) 7)
   ((b f) 3)
   (otherwise 9)
  )
3
```

The otherwise clause at the end means that if x is not a, b, d, e, or f, the case statement will return 9.

Iteration

The simplest iteration construct in Lisp is loop: a loop construct repeatedly executes its body until it hits a return special form. For example,

```
> (setq a 4)
4
> (loop
   (setq a (+ a 1))
   (when (> a 7) (return a))
  )
8
> (loop
   (setq a (- a 1))
   (when (< a 3) (return))
  )
nil
```

The next simplest is dolist: dolist binds a variable to the elements of a list in order and stops when it hits the end of the list.

```
> (dolist (x '(a b c)) (print x))
A
B
C
nil
```

Dolist always returns nil. Note that the value of x in the above example was never nil: the nil below the C was the value that dolist returned, printed by the read-eval-print loop.

The most complicated iteration primitive is called do. A do statement looks like this:

```
> (do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 5) y)
    (print y)
    (print 'working)
  )
1
WORKING
2
WORKING
4
WORKING
8
WORKING
16
WORKING
32
```

The first part of a do specifies what variables to bind, what their initial values are, and how to update them. The second part specifies a termination condition and a return value. The last part is the body. A do form binds its variables to their initial values like a let, then checks the termination condition. As long as the condition is false, it executes the body repeatedly; when the condition becomes true, it returns the value of the return-value form.

The do* form is to do as let* is to let.

Non-local Exits

The return special form mentioned in the section on iteration is an example of a nonlocal return. Another example is the return-from form, which returns a value from the surrounding function:

```
> (defun foo (x)
    (return-from foo 3)
    x
  )
FOO
> (foo 17)
3
```

Actually, the return-from form can return from any named block -- it's just that functions are the only blocks which are named by default. You can create a named block with the block special form:

```
> (block foo
    (return-from foo 7)
    3
  )
7
```

The return special form can return from any block named nil. Loops are by default labeled nil, but you can make your own nil-labeled blocks:

```
> (block nil
    (return 7)
    3
  )
7
```

Another form which causes a nonlocal exit is the error form:

```
> (error "This is an error")
Error: This is an error
```

The error form applies format to its arguments, then places you in the debugger.

Funcall, Apply, and Mapcar

Earlier I promised to give some functions which take functions as arguments. Here they are:

```
> (funcall #'+ 3 4)
7
> (apply #'+ 3 4 '(3 4))
14
> (mapcar #'not '(t nil t nil t nil))
(NIL T NIL T NIL T)
```

Funcall calls its first argument on its remaining arguments.

Apply is just like funcall, except that its final argument should be a list; the elements of that list are treated as if they were additional arguments to a funcall.

The first argument to mapcar must be a function of one argument; mapcar applies this function to each element of a list and collects the results in another list.

Funcall and apply are chiefly useful when their first argument is a variable. For instance, a search engine could take a heuristic function as a parameter and use funcall or apply to call that function on a state description. The sorting functions described later use funcall to call their comparison functions.

Mapcar, along with nameless functions (see below), can replace many loops.

Lambda

If you just want to create a temporary function and don't want to bother giving it a name, lambda is what you need.

```
> #'(lambda (x) (+ x 3))
#<Closure: #88b71ece>
> (funcall * 5)
8
```

The combination of lambda and mapcar can replace many loops. For example, the following two forms are equivalent:

```
> (do ((x '(1 2 3 4 5) (cdr x))
       (y nil))
      ((null x) (reverse y))
    (push (+ (car x) 2) y)
  )
(3 4 5 6 7)
> (mapcar #'(lambda (x) (+ x 2)) '(1 2 3 4 5))
(3 4 5 6 7)
```

Sorting

Lisp provides two primitives for sorting: sort and stable-sort.

```
> (sort '(2 1 5 4 6) #'<)
(1 2 4 5 6)
> (sort '(2 1 5 4 6) #'>)
(6 5 4 2 1)
```

The first argument to sort is a list; the second is a comparison function. The sort function does not guarantee stability: if there are two elements a and b such that (and (not (< a b)) (not (< b a))), sort may arrange them in either order. The stable-sort function is exactly like sort, except that it guarantees that two equivalent elements appear in the sorted list in the same order that they appeared in the original list.

Be careful: sort is allowed to destroy its argument, so if the original sequence is important to you, make a copy with the copy-list or copy-seq function.

Equality

Lisp has many different ideas of equality. Numerical equality is denoted by =. Two symbols are eq if and only if they are identical. Two copies of the same list are not eq, but they are equal.

```
> (eq 'a 'a)
T
> (eq 'a 'b)
NIL
> (= 3 4)
T
> (eq '(a b c) '(a b c))
NIL
> (equal '(a b c) '(a b c))
T
> (eql 'a 'a)
T
> (eql 3 3)
T
> (equalp 3 3.0)
T
```

The eql predicate is equivalent to eq for symbols and to = for numbers.

The equal predicate is equivalent to eql for symbols and numbers. It is true for two conses if and only if their cars are equal and their cdrs are equal. It is true for two structures if and only if the structures are the same type and their corresponding fields are equal.

The equalp predicate is the most flexible of all. It returns t for case insensitive characters and strings, numbers of different types, arrays, etc…

Some Useful List Functions

These functions all manipulate lists.

```
> (append '(1 2 3) '(4 5 6))    ;concatenate lists
(1 2 3 4 5 6)
> (reverse '(1 2 3))            ;reverse the elements of a list
(3 2 1)
> (member 'a '(b d a c))        ;set membership -- returns the first tail
(A C)                           ;whose car is the desired element
> (find 'a '(b d a c))          ;another way to do set membership
A
> (find '(a b) '((a d) (a d e) (a b d e) ()) :test #'subsetp)
(A B D E)                       ;find is more flexible though
> (subsetp '(a b) '(a d e))     ;set containment
nil
> (intersection '(a b c) '(b))  ;set intersection
(B)
> (union '(a) '(b))             ;set union
(A B)
> (set-difference '(a b) '(a))  ;set difference
(B)
```

Subsetp, intersection, union, and set-difference all assume that each argument contains no duplicate elements -- (subsetp '(a a) '(a b b)) is allowed to fail, for example.

Find, subsetp, intersection, union, and set-difference can all take a :test keyword argument; by default, they all use eql.

XLISP COMMAND LOOP

When XLISP is started, it first tries to load the workspace "xlisp.wks", or an alternative file specified with the "-w*filename*" option, from the current directory. If that file doesn't exist, or the "-w" flag is in the command line, XLISP builds an initial workspace, empty except for the built-in functions and symbols.

Then, providing no workspace file was loaded, XLISP attempts to load "init.lsp" from a path in XLPATH or the current directory. This file can be modified to suit the user's requirements. It contains a number of preference items.

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

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

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

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

```
>
```

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

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

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

Backspace	delete last character
Del	delete last character
tab	tabs over (treated as space by XLISP reader)
ctrl-C	goto top level
ctrl-G	cleanup and return one level
ctrl-Z	On Windows - end of file (returns one level or exits program)
ctrl-D	On OS X/Linux/Unix – end of file (returns one level or exits program)
ctrl-P	proceed (continue)
ctrl-T	print information

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

ctrl-B	BREAK -- enter break loop
ctrl-S	Pause until another key is struck
ctrl-C	go to top level
ctrl-T	print information

Under many operating systems (Linux, OS X, MS-DOS) if the global variable **dos-input** is set non-nil, the C language line editor or the operating system's line editor in the case of MS-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 EMACS or 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. If the environment variables EMACS or EPSRUN are defined then **dos-input** is automatically set to *T*.

If the global variable **dos-input** is nil, or in the Windows version, a special internal line editor is used. In this case the last 20 lines are saved, and can be recalled and viewed using the up and down arrow keys. Duplicate lines are not saved.

An additional feature is symbol name lookup. This command takes what appears to be an incomplete symbol name to the left of the cursor and prints all interned symbol names that match. Only names with function bindings will match immediately after a left parenthesis (for the Linux and OS X versions of XLISP).

The control keys for the editor are:

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

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

BREAK COMMAND LOOP

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

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

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

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

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

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

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

DATA TYPES

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

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

- NIL
  Unlike the original XLISP, NIL is a symbol (although not in the *obarray*), to allowing setting its properties.

- lists
  Either NIL or a CDR-linked list of cons cells, terminated by a symbol (typically NIL). Circular lists are allowable, but can cause problems with some functions so they must be used with care.

- arrays
  The CDR field of an array points to the dynamically allocated data array, while the CAR contains the integer length of the array. Elements in the data array are pointers to other cells [Size limited to about 16360].

- character strings
  Implemented like arrays, except string array is byte indexed and contains the actual characters. Note: unlike the underlying C, the null character (value 0) is valid. [Size limited to about 65500]

- symbols
  Implemented as a 4 element array. The elements are value cell, function cell, property list, and print name (a character string node). Print names are limited to 100 characters. There are also flags for constant and special. Values bound to special symbols (declared with DEFVAR or DEFPARAMETER) are always dynamically bound, rather than being lexically bound.

- fixnums (integers)
  Small integers (> -129 and <256) are statically allocated and are thus always EQ integers of the same value. The CAR field is used to hold the value, which is a 32 bit signed integer.

- bignums (integers)
  Big integers which don't fit in fixnums are stored in a special structure. Part of the bignum extension compilation option, when absent fixnums will overflow into flonums. Fixnums and flonums are collectively referred to as *integers*. [size limit is about 65500 characters for printing or about 500000 bits for calculations].

- ratios
  The CAR field is used to hold the numerator while the CDR field is used to hold the denominator. The numerator and denominator are stored as either both bignums or both fixnums. All ratios results are returned in reduced form, and are returned as integers if the denominator is 1. Part of the bignum extension. Ratios and integers are collectively referred to as *rationals*.

- characters
  All characters are statically allocated and are thus EQ characters of the same value. The CAR field is used to hold the value. In XLISP characters are unsigned and thus range in value from 0 to 255.

- flonums (floating point numbers)
  The CAR and CDR fields hold the value, which is typically a 64 bit IEEE floating point number. Flonums and rational numbers are collectively referred to as real numbers.

- complex numbers
  Part of the math extension compilation option. The CAR field is used to hold the real part while the CDR field is used to hold the imaginary part. The parts can be either both rationals (ratio or integer) or both flonums. Any function which would return an integer complex number with a zero imaginary part returns just the real integer.

- objects
  Implemented as an array of instance variable count plus one elements. The first element is the object's class, while the remaining arguments are the instance variables.

- streams (file)
  The CAR and CDR fields are used in a system dependent way as a file pointer.

- streams (unnamed -- string)
  Implemented as a tconc-style list of characters.

- subrs (built-in functions)
  The CAR field points to the actual code to execute, while the CDR field is an internal pointer to the name of the function.

- fsubrs (special forms)
  Same implementation as subrs.

- closures (user defined functions)
  Implemented as an array of 11 elements:
  1. name symbol or NIL
  2. 'lambda or 'macro
  3. list of required arguments
  4. optional arguments as list of (<arg> <init> <specified-p>) triples.
  5. &rest argument
  6. &key arguments as list of (<key> <arg> <init> <specified-p>) quadruples.
  7. &aux arguments as list of (<arg> <init>) pairs.
  8. function body
  9. value environment (see page 107 for format)
  10. function environment
  11. argument list (unprocessed)

- structures
  Implemented as an array with first element being a pointer to the structure name string, and the remaining elements being the structure elements.

- hash-tables
  Implemented as a structure of varying length with no generalized accessing functions, but with a special print function (print functions not available for standard structures).

- random-states
  Implemented as a structure with a single element which is the random state (here a fixnum, but could change without impacting XLISP programs).

- packages
  Implemented using a structure. Packages must only be manipulated with the functions provided.

THE EVALUATOR

The process of evaluation in XLISP:

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

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

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

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

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

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

HOOK FUNCTIONS

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

If the symbol **evalhook** is bound to a function closure, then every call of eval will call this function. The function takes two arguments, 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 escape mechanism is provided -- execution of *top-level*, or any error that causes return to the top level, will unhook the functions. Applications should bind these values either via *progv*, *evalhook*, or *applyhook*.

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

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

LEXICAL CONVENTIONS

The following conventions must be followed when entering XLISP programs:

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

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

( ) ' ` , " ;

and the escape characters:

\ |

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

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

For semantic reasons, certain character sequences should/can never be used as symbols in XLISP. A single period is used to denote dotted lists. The symbol t is also reserved for use as the truth value. The symbol nil represents an empty list.

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

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

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

Flonum (floating point) literals consist of a sequence of digits optionally beginning with a sign ("+" or "-") and including one or both of an embedded (not trailing) decimal point or a trailing exponent. The optional exponent is denoted by an "E" or "e" followed by an optional sign and one or more digits. The range of values a floating point number can represent is limited by the size of a C "double" on most machines on which XLISP is running.

Numeric literals cannot have embedded escape characters. If they do, they are treated as symbols. Thus "12\3" is a symbol even though it would appear to be identical to '123'. Conversely, symbols that could be interpreted as numeric literals in the current radix must have escape characters.

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

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

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

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

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

8 BIT ASCII CHARACTERS

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

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

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

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

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

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

READTABLES

The behavior 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 occurrence of the specified value. Of course, the readmacro code is free to read additional characters from the input stream. A :nmacro is a symbol constituent except as the first character of a symbol.

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

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

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

XLISP defines several useful read macros:

'<expr> == (quote <expr>)
`<expr> == (backquote <expr>)
,<expr> == (comma <expr>)
,@<expr> == (comma-at <expr>)
#'<expr> == (function <expr>)
#(<expr>…)== an array of the specified expressions
#S(<structtype> [<slotname> <value>]…)
== structure of specified type and initial values
#.<expr> == result of evaluating <expr>
#d<digits> == a decimal number (integer or ratio)
#x<hdigits> == a hexadecimal integer or ratio (0-9,A-F)
#o<odigits> == an octal integer or ratio (0-7)
#b<bdigits> == a binary integer or ratio (0-1)
#<base>r<digits>
== an integer or ratio in base <base>, 2-36
#| |# == a comment
#:<symbol> == an uninterned symbol
#C(r i) == a complex number
#+<expr> == conditional on feature expression true
#-<expr> == conditional on feature expression false

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

SYMBOL CASE CONTROL

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

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

The printing of symbols involves the settings of both *READTABLE-CASE* and *PRINT-CASE*. When *READTABLE-CASE* is :UPCASE, lowercase characters are escaped (unless PRINC is used), and uppercase characters are printed in the case specified by *PRINT-CASE*. When *READTABLE-CASE* is :DOWNCASE, uppercase characters are escaped (unless PRINC is used), and lowercase are printed in the case specified by *PRINT-CASE*. The *PRINT-CASE* value of :CAPITALIZE means that the first character of the symbol, and any character in the symbol immediately following a non-alphabetical character are to be in uppercase, while all other alphabetical characters are to be in lowercase. The remaining *READTABLE-CASE* modes ignore *PRINT-CASE* and do not escape alphabetic characters. :PRESERVE never changes the case of characters while :INVERT inverts the case of any non mixed-case symbols.

There are five major useful combinations of these modes:

A: *READTABLE-CASE* :UPCASE *PRINT-CASE* :UPCASE
"Traditional" mode. Case insensitive input; must escape to put lowercase characters in symbol names. Symbols print exactly as they are stored, with lowercase characters escaped when PRIN1 is used.

B: *READTABLE-CASE* :UPCASE *PRINT-CASE* :DOWNCASE
"Eyesaver" mode. Case insensitive input; must escape to put lowercase characters in symbol name. Symbols print entirely in lowercase except symbols escaped when lowercase characters present with PRIN1.

C: *READTABLE-CASE* :PRESERVE
"Oldfashioned case sensitive" mode. Case sensitive input. Predefined symbols must be typed in uppercase. No alpha quoting needed. Symbols print exactly as stored.

D: *READTABLE-CASE* :INVERT
"Modern case sensitive" mode. Case sensitive input. Predefined symbols must be typed in lowercase. Alpha quoting should be avoided. Predefined symbols print in lower case, other symbols print as they were entered.

E: *READTABLE-CASE* :UPCASE *PRINT-CASE* :CAPITALIZE
Like case B, except symbol names print capitalized.

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

PACKAGES

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

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

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

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

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

For the subtleties of the package facility, read the section called 'Living with Packages' in page 126, and 'Common Lisp the Language', second edition.

LAMBDA LISTS

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

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

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

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

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

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

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

Here is the complete syntax for lambda lists:

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

where:

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

GENERALIZED VARIABLES

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

XLISP has a number of place forms pre-defined in code. In addition, the function DEFSETF can be used to define new place forms. Place forms and functions that take them as arguments must be carefully defined so that no expression is evaluated more than once, and evaluation proceeds from left to right. The result of this is that these functions tend to be slow. If multiple evaluation and execution order is not a concern, alternative simpler functions can be un-commented in common.lsp and common2.lsp.

A place form may be one of the following:

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

OBJECTS

Definitions:

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

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

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

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

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

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

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

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

The 'Object' CLASS

Object THE TOP OF THE CLASS HIERARCHY

Messages:

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

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

:prin1 [<stream>] PRINT THE OBJECT
<stream> t is *terminal-io*, (default, or nil, is *standard-output*)
returns the object

:isnew THE DEFAULT OBJECT INITIALIZATION ROUTINE
returns the object

:superclass GET THE SUPERCLASS OF THE OBJECT
Defined in classes.lsp. (See :superclass below)
returns nil

:ismemberof <class> CLASS MEMBERSHIP
Defined in classes.lsp
<class> class name
returns t if object member of class, else nil

:iskindof <class> CLASS MEMBERSHIP
Defined in classes.lsp
<class> class name
returns t if object member of class or subclass of class, else nil

:respondsto <sel> SELECTOR KNOWLEDGE
Defined in classes.lsp
<sel> message selector
returns t if object responds to message selector, else nil

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

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
- Defined in classes.lsp
- returns — the superclass (of the class)

:messages — GET THE LIST OF MESSAGES OF THE CLASS
- Defined in classes.lsp
- returns — association list of message selectors and closures for messages

:storeon — READ REPRESENTATION
- Defined in classes.lsp
- returns — a list, that when executed will re-create the class and its methods

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

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

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

INSTANCE VARIABLES OF CLASS 'CLASS':

MESSAGES - An association list of message names and closures implementing the messages.
IVARS - List of names of instance variables.
CVARS - List of names of class variables.
CVAL - Array of class variable values.
SUPERCLASS - The superclass of this class or nil if no superclass (only for class OBJECT).
IVARCNT - instance variables in this class (length of IVARS)
IVARTOTAL - total instance variables for this class and all superclasses of this class.
PNAME - printname string for this class.

SYMBOLS

All values are initially nil unless otherwise specified. All are special variables unless indicated to be constants.

- NIL - represents empty list and the Boolean value for "false". The value of NIL is nil, and cannot be changed (it is a constant). (car NIL) and (cdr NIL) are also defined to be nil.
- T - Boolean value "true" is constant with value t.
- self - within a method context, the current object (see page 36), otherwise initially unbound.
- object - constant, value is the class 'Object.'
- class - constant, value is the class 'Class'.
- internal-time-units-per-second - integer constant to divide returned times by to get time in seconds.
- pi - floating point approximation of pi (constant defined when math extension is compiled).
- version – constant cons for which the car is the major version number, 3, and the cdr is the minor version number, 5 at the time of this writing. Installations earlier than 3.05 will create this constant and set to (3 . 4).
- char-code-limit - a constant defined in common2.lsp indicating the exclusive upper limit of the values returned by the function char-code. Its value is 128, which means that char-code will return values in the range 0 .. 127.
- *obarray* - the object hash table. Length of array is a compilation option. Objects are hashed using the hash function and are placed on a list in the appropriate array slot. This variable does not exist when the package feature is compiled in.
- *package* - the current package. Do not alter. Part of the package feature.
- *modules* - a list of names of the modules loaded so far. It is used by the functions provide and require.
- *terminal-io* - stream bound to keyboard and display. Do not alter.
- *standard-input* - the standard input stream, initially stdin. If stdin is not redirected on the command line, then *terminal-io* is used so that all interactive I/O uses the same stream.
- *standard-output* - the standard output stream, initially stdout. If stdout is not redirected on the command line then *terminal-io* is used so that all interactive I/O uses the same stream.
- *error-output* - the error output stream (used by all error messages), initially same as *terminal-io*.
- *trace-output* - the trace output stream (used by the trace function), initially same as *terminal-io*.
- *debug-io* - the break loop I/O stream, initially same as *terminal-io*. System messages (other than error messages) also print out on this stream.
- *breakenable* - flag controlling entering break loop on errors (see page 22)
- *tracelist* - list of names of functions to trace, as set by trace function.
- *tracenable* - enable trace back printout on errors (see page 22).
- *tracelimit* - number of levels of trace back information (see page 22).
- *evalhook* - user substitute for the evaluator function (see page 26, and evalhook and applyhook functions).
- *applyhook* - user substitute for function application (see page 26, and evalhook and applyhook functions).
- *readtable* - the current readtable (see page 30).
- *gc-flag* - controls the printing of gc messages. When non-nil, a message is printed after each garbage collection giving the total number of nodes and the number of nodes free.
- *gc-hook* - function to call after garbage collection (see page 26).

- *integer-format* - format for printing integers (when not bound to a string, defaults to "%d" or "%ld" depending on implementation). Variable not used when bignum extension installed.
- *float-format* - format for printing floats (when not bound to a string, defaults to "%g")
- *readtable-case* - symbol read and output case. See page 32 for details
- *read-base* - When bound to a fixnum in the range 2 through 36, determines the default radix used when reading rational numbers. Part of bignum extension.
- *print-base* - When bound to a fixnum in the range 2 through 36, determines the radix used when printing rational numbers with prin1 and princ. Part of bignum extension.
- *print-case* - symbol output case when printing. See page 32 for details
- *print-level* - When bound to a number, list levels beyond this value are printed as '#'. Used by all printing functions. Good precaution to avoid getting caught in circular lists.
- *print-length* - When bound to a number, lists longer than this value are printed as '…'. Used by all printing functions. Good precaution to avoid getting caught in circular lists.
- *dos-input* - When not nil, uses DOS line input function for read (see page 20).
- *displace-macros* - When not nil, macros are replaced by their expansions when executed (see page 25).
- *random-state* - the default random-state used by the random function.
- *features* - list of features, initially (:xlisp), used for #+ and #- reader macros.
- *startup-functions* - list of functions to be executed when workspace started
- *command-line* - the XLISP command line, in the form of a list of strings, one string per argument.
- *load-file-arguments* - When not nil, file arguments are loaded at startup.
- *top-level-loop* - Top level loop to utilize, defaults to TOP-LEVEL-LOOP. Note that this function can only be restarted by executing TOP-LEVEL, and it never exits.
- *read-suppress* - When not nil, inhibits certain parts of reading. Used by the #+ and #- macros.
- *gensym-counter* - Unsigned integer suffix used by the GENSYM function.

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

EVALUATION FUNCTIONS

(eval <expr>) EVALUATE AN XLISP EXPRESSION

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

(apply <fun> <arg>…<args>) APPLY A FUNCTION TO A LIST OF ARGUMENTS

<fun> the function to apply (or function symbol). May not be macro or fsubr.
<arg> initial arguments, which are CONSed to…
<args> the argument list
returns the result of applying the function to the arguments

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

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

(quote <expr>) RETURN AN EXPRESSION UNEVALUATED

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

(function <expr>) GET THE FUNCTIONAL INTERPRETATION

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

(complement <fun>) MAKE A COMPLEMENTARY FUNCTION

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

(identity <expr>) RETURN THE EXPRESSION

<expr> the expression
returns the expression

(backquote <expr>) FILL IN A TEMPLATE

fsubr. Note: an improved backquote facility, which works properly when nested, is available by loading the file backquot.lsp.
<expr> the template (quoted)
returns a copy of the template with comma and comma-at expressions expanded

(comma <expr>) COMMA EXPRESSION

returns (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

returns (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. See page 34 for a full description of the argument list.

<args> formal argument list (lambda list) (quoted)

<expr> expressions of the function body (quoted)

returns the function closure

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

<closure> the closure

returns the original lambda expression, or nil if not a closure. Second return value is t if closure has a non-global environment, and the third return value is the name of the closure.

(macroexpand <form>) RECURSIVELY EXPAND MACRO CALLS

<form> the form to expand

returns the macro expansion

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

<form> the macro call form

returns the macro expansion

MULTIPLE VALUE FUNCTIONS

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

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

Defined as macro in common.lsp.

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

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

fsubr.

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

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

Defined as macro in common.lsp.

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

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

fsubr.

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

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

Defined as macro in common.lsp.

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

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

fsubr.

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

(values [<expr>]) RETURN MULTIPLE VALUES

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

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

Defined in common.lsp.

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

SYMBOL FUNCTIONS

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

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

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

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

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

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

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

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

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

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

fsubr. See also the section starting on page 132.

<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

By default the symbol is named “G” followed by an incrementing integer count.

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

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

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

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

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

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

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

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

May be used as a place form.

<sym>	the symbol
returns	the symbol's value

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

May be used as a place form.

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

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

May be used as a place form.

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

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

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

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

You cannot unbind constants.

<sym>	the symbol
returns	the symbol

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

<sym>	the symbol
returns	the symbol

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

Defined in common.lsp if package extension not compiled.

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

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

fsubr.

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

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

fsubr.

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

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

fsubr. Variable only initialized if not previously defined.

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

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

Also see definition of PROCLAIM and DECLARE.

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

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

Defined as macro in common.lsp, and provided to assist in porting Common Lisp applications to XLISP-PLUS.

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

(proclaim <proc>) PROCLAIM GLOBAL SYMBOL ATTRIBUTES

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

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

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

Defined in common2.lsp.

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

GENERALIZED VARIABLE FUNCTIONS

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

fsubr.

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

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

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

<place>	the field specifier
<expr>	the new value
returns	nil

(defsetf <sym> <fcn> [<doc>]) DEFINE A SETF FIELD SPECIFIER
(defsetf <sym> <fargs> (<value>) <expr>…)

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

<sym>	field specifier symbol (quoted)
<fcn>	function to use (quoted symbol) which takes the same arguments as the field specifier plus an additional argument for the value. The value must be returned.
<doc>	optional documentation string for GLOS and DOCUMENTATION
<fargs>	formal argument list of unevaluated arguments (lambda list) (quoted).
<value>	symbol bound to value to store (quoted)
<expr>	The last expression must an expression to evaluate in the setf context. In this respect, defsetf works like a macro definition.
returns	the field specifier symbol

(push <expr> <place>) CONS TO A FIELD

Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-nil for best performance.

<place>	field specifier being modified (see setf)
<expr>	value to cons to field
returns	the new value which is (CONS <expr> <place>)

(pushnew <expr> <place> &key :test :test-not :key) CONS NEW TO A FIELD

Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-nil for best performance.

<place>	field specifier being modified (see setf)
<expr>	value to cons to field, if not already MEMBER of field
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function list argument (defaults to identity)
returns	the new value which is (CONS <expr> <place>) or <place>

(pop <place>) REMOVE FIRST ELEMENT OF A FIELD

Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-nil for best performance.

<place> the field being modified (see setf)
returns (CAR <place>), field changed to (CDR <place>)

(incf <place> [<value>]) INCREMENT A FIELD
(decf <place> [<value>]) DECREMENT A FIELD

Defined as macro in common.lsp. Only evaluates place form arguments one time. It is recommended that *displace-macros* be non-nil for best performance.

<place> field specifier being modified (see setf)
<value> Numeric value (default is 1)
returns the new value which is (+ <place> <value>) or (- <place> <value>)

PACKAGE FUNCTIONS

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

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

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

Defined in common.lsp.

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

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

Defined as macro in common.lsp. Use to define a package, or redefine a package.

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

(delete-package <package>) DELETE A PACKAGE

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

<package>	package to delete
returns	t if successful

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

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

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

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

<symbols>	symbols to declare as external
<package>	package symbol is in
returns	t

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

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

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

<package>	package to find
returns	package with name or nickname <package>, or nil if not found

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

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

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

<symbols>	symbols to import (fully qualified names)
<package>	package to import symbols into
returns	t

(in-package <package>) SET CURRENT PACKAGE

fsubr. Sets the current package until next call or end of current LOAD.

<package>	the package to enter
returns	the package

(list-all-packages) GET ALL PACKAGE NAMES

returns	list of all currently existing packages

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

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

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

<package> package name
returns package name string

(package-nicknames <package>) GET PACKAGE NICKNAME STRINGS

<package> package name
returns list of package nickname strings

(package-obarray <package> [<external>]) GET AN OBARRAY

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

(package-shadowing-symbols <package>) GET LIST OF SHADOWING SYMBOLS

<package> the package
returns list of shadowing symbols in package

(package-use-list <package>) GET PACKAGES USED BY A PACKAGE

<package> the package
returns list of packages used by this package (as in USE-PACKAGE)

(package-used-by-list <package>)
GET PACKAGES THAT USE THIS PACKAGE

<package> the package
returns list of packages that use this package (as in USE-PACKAGE)

(package-valid-p <package>) IS THIS A GOOD PACKAGE?

<package> object to check
returns t if a valid package, else nil

(provide <name>) ADD NEW MODULE TO LIST OF MODULES

Defined in common.lsp.
<name> a string or a symbol identifying the new module
returns the list of all provided modules

(rename-package <package> <new> [<nick>]) RENAME A PACKAGE

<package> original package
<new> new package name (may be same as original name)
<nick> list of new package nicknames
returns the new package

(require <name>[<path>…]) LOAD NEW MODULE

Defined in common.lsp.
<name> a string or symbol identifying the module required (case sensitive)
<path> the pathname(s) of file(s) to load if module isn't already provided. If no pathnames are given, attempts to load file of the same name with the path specified by the OS variable XLPATH.
returns NIL

(shadow <symbols> [<package>]) MAKE SHADOWING SYMBOLS

If a symbol is not already in the package, it is interned. The symbol is placed in the shadowing symbols list for the package.

<symbols> the symbol or symbols to shadow
<package> package to put symbols in
returns t

(shadowing-import <symbols> [<package>])
IMPORT SYMBOLS AND SHADOW

If a symbol exists in the package, it is first uninterned. The symbol is imported, and then made shadowing.

<symbols> the symbol or symbols to import and shadow
<package> package to put symbols in
returns t

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

<symbol> the symbol
returns the home package of the symbol, or nil if none

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

<symbols> symbol or symbols to make internal
<package> package for symbols
returns t

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

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

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

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

PROPERTY LIST FUNCTIONS

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

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

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

<sym>	the symbol
<prop>	the property name
<dflt>	value to return if property not found, (default is nil)
returns	the property value or <dflt> if property doesn't exist

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

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

<place>	where the property list is stored
<prop>	the property name
<dflt>	value to return if property not found, (default is nil)
returns	the property value or <dflt> if property doesn't exist

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

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

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

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

Defined as a macro in common.lsp.

<place>	where the property list is stored
<prop>	the property name
returns	t if property existed, else nil

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

<sym>	the symbol
<prop>	the property name
returns	nil

HASH TABLE FUNCTIONS

A hash table is implemented as an structure of type hash-table. No general accessing functions are provided, and hash tables print out using the angle bracket convention (not readable by READ). The first element is the comparison function. The remaining elements contain association lists of keys (that hash to the same value) and their data. See also the function hash on page 45.

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

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

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

May be used as a place form.

<key>	hash key
<table>	hash table
<def>	value to return on no match (default is nil)
returns	two values: the first is the associated data, if found, or <def> if not found. The second is t if found or nil otherwise.

(remhash <key> <table>) DELETE FROM HASH TABLE

<key>	hash key
<table>	hash table
returns	t if deleted, nil if not in table

(clrhash <table>) CLEAR THE HASH TABLE

<table>	hash table
returns	nil, all entries cleared from table

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

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

(maphash <fcn> <table>) MAP FUNCTION OVER TABLE ENTRIES

<fcn>	the function or function name, a function of two arguments, the first is bound to the key, and the second the value of each table entry in turn.
<table>	hash table
returns	nil

SEQUENCE FUNCTIONS

These functions work on sequences -- lists, arrays, or strings.

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

If result type is string, sequences must contain only characters.

<type>	result type, one of CONS, LIST, ARRAY, or STRING
<expr>	zero or more sequences to concatenate
returns	a sequence which is the concatenation of the argument sequences

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

May be used as a place form

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

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

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

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

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

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

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

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

Note: a circular list causes an error. To detect a circular list, use LIST-LENGTH.

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

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

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

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

<seq>	the sequence
<start>	the starting position (zero origin)
<end>	the ending position + 1 (default, or nil, is end of sequence)
returns	the sequence between <start> and <end>

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

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

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

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

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

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

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

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

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

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

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

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

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

DELETE ELEMENTS FROM A SEQUENCE

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

DELETE ELEMENTS THAT PASS TEST

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

DELETE ELEMENTS THAT FAIL TEST

Note: These are the destructive versions of remove, remove-if and remove-if-not, respectively.

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

(substitute <r> <e> <s> &key :key :test :test-not :start :end :count :from-end)

SUBSTITUTE ELEMENTS IN A SEQUENCE

(substitute-if <r> <test> <s> &key :key :start :end :count :from-end)

SUBSTITUTE ELEMENTS THAT PASS TEST

(substitute-if-not <r> <test> <s> &key :key :start :end :count :from-end)

SUBSTITUTE ELEMENTS THAT FAIL TEST

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

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

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

(reduce <fcn> <seq> &key :initial-value :start :end) REDUCE SEQUENCE TO SINGLE VALUE

<fcn>	function (of two arguments) to apply to result of previous function application (or first element) and each member of sequence
<seq>	the sequence
:initial-value	value to use as first argument in first function application rather than using the first element of the sequence
:start	starting index
:end	index of end+1, or nil for (length <seq>)
returns	if sequence is empty and there is no initial value, returns result of applying function to zero arguments. If there is a single element, returns the element. Otherwise returns the result of the last function application.

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

Delete-duplicates defined in common2.lsp.

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

(fill <seq> <expr> &key :start :end) REPLACE ITEMS IN SEQUENCE

Defined in common.lsp.

<seq>	the sequence
<expr>	new value to place in sequence
:start	starting index
:end	index of end+1, or nil for (length <seq>)
returns	sequence with items replaced with new item

(replace <seq1> <seq2> &key :start1 :end1 :start2 :end2)
REPLACE ITEMS IN SEQUENCE FROM SEQUENCE

Defined in common.lsp.

<seq1>	the sequence to modify
<seq2>	sequence with new items
:start1	starting index in <seq1>
:end1	index of end+1 in <seq1> or nil for end of sequence
:start2	starting index in <seq2>
:end2	index of end+1 in <seq2> or nil for end of sequence
returns	first sequence with items replaced

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

Defined in common2.lsp.

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

(copy-seq <seq>) COPY A SEQUENCE

Defined in common2.lsp.

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

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

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

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

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

FIND DIFFERENCE BETWEEN TWO SEQUENCES

Defined in common2.lsp.

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

LIST FUNCTIONS

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

May be used as a place form.

<expr> the list node

returns the car of the list node

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

May be used as a place form.

<expr> the list node

returns the cdr of the list node

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

May be used as place forms when common2.lsp loaded. fifth through tenth defined in common2.lsp.

returns the desired element(s) of the list

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

<expr1> the car of the new list node

<expr2> the cdr of the new list node

returns the new list node

(acons <expr1> <expr2> <alist>) ADD TO FRONT OF ASSOC LIST

Defined in common.lsp.

<expr1> key of new association

<expr2> value of new association

<alist> association list

returns new association list, which is (cons (cons <expr1> <expr2>) <expr3>))

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

<expr> expressions to be combined into a list

returns the new list

(append <expr>…) APPEND LISTS

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

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

Defined in common2.lsp.

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

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

<list>	the list
returns	the length of the list or nil if the list is circular

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

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

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

Defined in common2.lsp.

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

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

nbutlast defined in common2.lsp.

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

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

May be used as a place form.

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

(nthcdr <n> <list>) RETURN THE NTH CDR OF A LIST

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

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

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

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

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

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

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

(mapc <fcn> <list1> <list>…) APPLY FUNCTION TO SUCCESSIVE CARS
(mapcar <fcn> <list1> <list>…) APPLY FUNCTION TO SUCCESSIVE CARS
(mapcan <fcn> <list1> <list>…) APPLY FUNCTION TO SUCCESSIVE CARS
(mapl <fcn> <list1> <list>…) APPLY FUNCTION TO SUCCESSIVE CDRS
(maplist <fcn> <list1> <list>…) APPLY FUNCTION TO SUCCESSIVE CDRS
(mapcon <fcn> <list1> <list>…) APPLY FUNCTION TO SUCCESSIVE CDRS

<fcn>	the function or function name
<listn>	a list for each argument of the function
returns	the first list of arguments (mapc or mapl), a list of the values returned (mapcar or maplist), or list of returned values nconc'd together (mapcan or mapcon)

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

subst does minimum copying as required by Common Lisp. nsubst is the destructive version.

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

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

sublis does minimum copying as required by Common Lisp. nsublis is the destructive version.

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

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

Defined in common.lsp.

<keys>	list of association keys
<values>	list of association values, same length as keys
<alist>	existing association list (default is nil)
returns	new association list

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

Defined in common2.lsp.

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

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

Defined in common.lsp.

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

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

Defined in common.lsp.

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

(copy-tree <tree>) COPY A TREE

Defined in common.lsp.

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

(intersection <list1> <list2> &key :test :test-not :key) SET FUNCTIONS
(union <list1> <list2> &key :test :test-not :key)
(set-difference <list1> <list2> &key :test :test-not :key)
(set-exclusive-or <list1> <list2> &key :test :test-not :key)
(nintersection <list1> <list2> &key :test :test-not :key)
(nunion <list1> <list2> &key :test :test-not :key)
(nset-difference <list1> <list2> &key :test :test-not :key)
(nset-exclusive-or <list1> <list2> &key :test :test-not :key)

set-exclusive-or and nset-exclusive-or defined in common.lsp. nunion, nintersection, and nset-difference are aliased to their non-destructive counterparts in common.lsp. "n" versions are potentially destructive.

<list1>	first list
<list2>	second list
:test	the test function (defaults to eql)
:test-not	the test function (sense inverted)
:key	function to apply to test function arguments (defaults to identity)
returns	intersection: list of all elements in both lists union: list of all elements in either list set-difference: list of all elements in first list but not in second list set-exclusive-or: list of all elements in only one list

(adjoin <expr> <list> :test :test-not :key) ADD UNIQUE TO LIST

<expr>	new element to add
<list>	the list
:test	the test function (defaults to eql)
:test-not	the test function <sense inverted)
:key	function to apply to test function arguments (defaults to identity)
returns	if element not in list then (cons <expr> <list>), else <list>

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

Defined in common2.lsp.

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

DESTRUCTIVE LIST FUNCTIONS

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

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

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

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

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

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

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

(nconc <list>…) DESTRUCTIVELY CONCATENATE LISTS

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

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

Defined in common2.lsp.

<list1>	first list
<list2>	second list
returns	second list concatenated to the end of the first list, which has been destructively reversed

ARRAY FUNCTIONS

Note that sequence functions also work on arrays.

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

May be used as a place form

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

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

<size>	the size of the new array (fixnum)
:initial-element	value to initialize all array elements (default is nil)
:initial-contents	sequence used to initialize all array elements, consecutive sequence elements are used for each array element. The length of the sequence must be the same as the size of the array
returns	the new array

(vector <expr>…) MAKE AN INITIALIZED VECTOR

<expr>	the vector elements
returns	the new vector

STRING FUNCTIONS

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

(string <expr>) MAKE A STRING FROM A SYMBOL, AN ASCII VALUE, OR A CHARACTER

<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 trimmed 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 trimmed 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 trimmed copy of the string

(string-upcase <str> &key :start :end) CONVERT TO UPPERCASE

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or nil for end of string
returns	a converted copy of the string

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

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or nil for end of string
returns	a converted copy of the string

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

<str>	the string
:start	the starting offset
:end	the ending offset + 1 or nil for end of string
returns	a converted copy of the string with each word having an initial uppercase letter and following lowercase letters

(nstring-upcase <str> &key :start :end) CONVERT TO UPPERCASE

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

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

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

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

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

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

Defined in common2.lsp.

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

(strcat <expr>…) CONCATENATE STRINGS

Defined as macro in init.lsp, and provided to maintain compatibility with XLISP.
See CONCATENATE for preferred function.

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

(string< <str1> <str2> &key :start1 :end1 :start2 :end2) COMPARE CASE SENSITIVE 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)

Note: case is significant with these comparison functions.

<str1>	the first string to compare
<str2>	the second string to compare
:start1	first substring starting offset
:end1	first substring ending offset + 1 or nil for end of string
:start2	second substring starting offset
:end2	second substring ending offset + 1 or nil for end of string
returns	string=: t if predicate is true, nil otherwise others: If predicate is true then number of initial matching characters, else nil

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

Note: case is not significant with these comparison functions -- all uppercase characters are converted to lowercase before being compared.

<str1>	the first string to compare
<str2>	the second string to compare
:start1	first substring starting offset
:end1	first substring ending offset + 1 or nil for end of string
:start2	second substring starting offset
:end2	second substring ending offset + 1 or nil for end of string
returns	string-equal: t if predicate is true, nil otherwise
	others: If predicate is true then number of initial matching characters, else nil

CHARACTER FUNCTIONS

(char <string> <index>) EXTRACT A CHARACTER FROM A STRING

<string>	the string
<index>	the string index (zero relative)
returns	the ASCII code of the character

(alphanumericp <chr>) IS THIS CHARACTER ALPHANUMERIC?

<chr>	the character
returns	true if the character is alphabetic or numeric, nil otherwise

(upper-case-p <chr>) IS THIS AN UPPER CASE CHARACTER?

<chr>	the character
returns	true if the character is upper case, nil otherwise

(lower-case-p <chr>) IS THIS A LOWER CASE CHARACTER?

<chr>	the character
returns	true if the character is lower case, nil otherwise

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

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

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

<chr>	the character
returns	true if the character is available in both cases, nil otherwise

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

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

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

<chr>	the character
returns	the ASCII character code (integer, parity bit stripped) in the range 0 .. char-code-limit -1 (see page 39)

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

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

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

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

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

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

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

<n> the digit weight (integer)
<radix> the radix (default is 10)
returns the digit character or nil

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

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

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

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

(character <expr>) CREATE A CHARACTER

Defined in common2.lsp.
<expr> single character symbol, string, or integer
returns <expr> converted into a character

(char-name <chr>) CHARACTER PRINT NAME

Defined in common2.lsp.
<chr> the character
returns string which is the name of the character, or nil if no name

(char< <chr1> <chr2>…) COMPARE CASE SENSITIVE CHARACTERS
(char<= <chr1> <chr2>…)
(char= <chr1> <chr2>…)
(char/= <chr1> <chr2>…)
(char>= <chr1> <chr2>…)
(char> <chr1> <chr2>…)

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

(char-lessp <chr1> <chr2>…) COMPARE CHARACTERS
(char-not-greaterp <chr1> <chr2>…)
(char-equal <chr1> <chr2>…)
(char-not-equal <chr1> <chr2>…)
(char-not-lessp <chr1> <chr2>…)
(char-greaterp <chr1> <chr2>…)

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

STRUCTURE FUNCTIONS

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

(defstruct name [<comment>] <slot-desc>…) CREATE A NEW STRUCTURE
or
(defstruct (name <option>…) [<comment>] <slot-desc>…)

fsubr.

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

The recognized options are:

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

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

Each slot description takes the form:

<name>

or:

(<name> <defexpr>)

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

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

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

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

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

For instance:

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

creates the following functions:

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

OBJECT FUNCTIONS

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

Messages defined for Object and Class are listed starting on page 37.

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

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

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

Valid only in method context

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

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

Defined as macro in classes.lsp.

<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
<doc>	documentation string

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

Methods for classes defined with defclass:

(send <object> :<ivar>)
Returns the specified instance variable

(send <object> :SET-IVAR <ivar> <value>)
Used to set an instance variable, typically with setf via (setf (send <object> :<ivar>) <value>)

(send <sym> :NEW {:<ivar> <init>})
Actually definition for :ISNEW. Creates new object initializing instance variables as specified in keyword arguments, or to their default if keyword argument is missing. Returns the object.

returns the new class object

(defmethod <class> <sym> [<doc>] <fargs> <expr> …) DEFINE A NEW METHOD

Defined as macro in classes.lsp.

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

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

Defined as macro in classes.lsp.

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

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

Defined in classes.lsp.

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

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

Defined in classes.lsp.

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

ARITHMETIC FUNCTIONS

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

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

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

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

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

Round, floor, and ceiling, and the second argument of truncate, are part of the math extension. Integers are returned as is.

<expr>	the real number
<denom>	real number to divide <expr> by before converting
returns	the integer result of converting the number, and, as a second return value, the remainder of the operation, defined as: expr - result × denom. The type is flonum if either argument is flonum, otherwise it is rational.

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

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

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

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

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

(+ [<expr>…]) ADD A LIST OF NUMBERS

With no arguments returns addition identity, 0 (integer)

<expr>	the numbers
returns	the result of the addition

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

<expr>	the numbers
returns	the result of the subtraction

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

With no arguments returns multiplication identity, 1

<expr> the numbers

returns the result of the multiplication

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

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

<expr> the numbers

returns the result of the division

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

<expr> the number

returns the number plus one

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

<expr> the number

returns the number minus one

(rem <expr>…) REMAINDER OF A LIST OF NUMBERS

With the math extension, only two arguments allowed.

<expr> the real numbers (must be integers, without math extension)

returns the result of the remainder operation (remainder with truncating division)

(mod <expr1> <expr2>) NUMBER MODULO ANOTHER NUMBER

Part of the math extension.

<expr1> real number

<expr2> real number divisor (may not be zero)

returns the remainder after dividing <expr1> by <expr2> using flooring division, thus there is no discontinuity in the function around zero.

(min <expr>…) THE SMALLEST OF A LIST OF NUMBERS

<expr> the real numbers

returns the smallest number in the list

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

<expr> the real numbers

returns the largest number in the list

(abs <expr>) THE ABSOLUTE VALUE OF A NUMBER

<expr> the number

returns the absolute value of the number, which is the floating point magnitude for complex numbers

(signum <expr>) GET THE SIGN OF A NUMBER

Defined in common.lsp.

<expr> the number

returns zero if number is zero, one if positive, or negative one if negative. Numeric type is same as number. For a complex number, returns unit magnitude but same phase as number.

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

Defined in common2.lsp.

<expr1>	the real number
<expr2>	another real number (default is 1.0)
returns	the number <expr2> with the sign of <expr1>

(gcd [<n>…]) COMPUTE THE GREATEST COMMON DIVISOR

With no arguments returns 0, with one argument returns the argument.

<n>	The number(s) (integer)
returns	the greatest common divisor

(lcm <n>…) COMPUTE THE LEAST COMMON MULTIPLE

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

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

(random <n> [<state>]) COMPUTE A PSEUDO-RANDOM NUMBER

<n>	the real number upper bound
<state>	a random-state (default is *random-state*)
returns	a random number in range [0,n)

(make-random-state [<state>]) CREATE A RANDOM-STATE

<state>	a random-state, t, or nil (default, or nil, is *random-state*)
returns	If <state> is t, a random random-state, otherwise a copy of <state>

(sin <expr>) COMPUTE THE SINE OF A NUMBER
(cos <expr>) COMPUTE THE COSINE OF A NUMBER
(tan <expr>) COMPUTE THE TANGENT OF A NUMBER
(asin <expr>) COMPUTE THE ARC SINE OF A NUMBER
(acos <expr>) COMPUTE THE ARC COSINE OF A NUMBER

<expr>	the floating point number
returns	the sine, cosine, tangent, arc sine, or arc cosine of the number

(atan <expr> [<expr2>]) COMPUTE THE ARC TANGENT OF A NUMBER

<expr>	the floating point number (numerator)
<expr2>	the denominator (default is 1). May only be specified if math extension installed
returns	the arc tangent of <expr>/<expr2>

(sinh <expr>) COMPUTE THE HYPERBOLIC SINE OF A NUMBER
(cosh <expr>) COMPUTE THE HYPERBOLIC COSINE OF A NUMBER
(tanh <expr>) COMPUTE THE HYPERBOLIC TANGENT OF A NUMBER
(asinh <expr>) COMPUTE THE HYPERBOLIC ARC SINE OF A NUMBER
(acosh <expr>) COMPUTE THE HYPERBOLIC ARC COSINE OF A NUMBER
(atanh <expr>) COMPUTE THE HYPERBOLIC ARC TANGENT OF A NUMBER

Defined in common.lsp.

<expr>	the number
returns	the hyperbolic sine, cosine, tangent, arc sine, arc cosine, or arc tangent of the number

(expt <x-expr> <y-expr>) COMPUTE X TO THE Y POWER

<x-expr> the number
<y-expr> the exponent
returns x to the y power. If y is an integer, then the result type is the same as the type of x.

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

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

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

Defined in common.lsp.
<x-expr> the number
returns e to the ix power

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

Part of the math extension.
<expr> the number
<base> the base (default is e)
returns log base <base> of <expr>

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

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

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

Defined in common2.lsp.
<expr> non-negative integer
returns the integer square root, either exact or the largest integer less than the exact value

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

Part of the bignum extension.
<expr> rational number
returns numerator of number (number if integer)

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

Part of the bignum extension.
<expr> rational number
returns denominator of number (1 if integer)

(complex <real> [<imag>]) CONVERT TO COMPLEX NUMBER

Part of the math extension.
<real> real number real part
<imag> real number imaginary part (default is 0)
returns the complex number

(realpart <expr>) GET THE REAL PART OF A NUMBER

Part of the math extension.
<expr> the number
returns the real part of a complex number, or the number itself if a real number

(imagpart <expr>) GET THE IMAGINARY PART OF A NUMBER

Part of the math extension.

<expr> the number

returns the imaginary part of a complex number, or zero of the type of the number if a real number

(conjugate <expr>) GET THE CONJUGATE OF A NUMBER

Part of the math extension.

<expr> the number

returns the conjugate of a complex number, or the number itself if a real number

(phase <expr>) GET THE PHASE OF A NUMBER

Part of the math extension.

<expr> the number

returns the phase angle, equivalent to (atan (imagpart <expr>) (realpart <expr>))

(< <n1> <n2>…) TEST FOR LESS THAN

(<= <n1> <n2>…) TEST FOR LESS THAN OR EQUAL TO

(= <n1> <n2>…) TEST FOR EQUAL TO

(/= <n1> <n2>…) TEST FOR NOT EQUAL TO

(>= <n1> <n2>…) TEST FOR GREATER THAN OR EQUAL TO

(> <n1> <n2>…) TEST FOR GREATER THAN

<n1> the first real number to compare

<n2> the second real number to compare

returns the result of comparing <n1> with <n2>

BITWISE LOGICAL FUNCTIONS

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

(logand [<expr>…]) THE BITWISE AND OF A LIST OF INTEGERS
With no arguments returns identity -1
<expr> the integers
returns the result of the and operation

(logior [<expr>…]) THE BITWISE INCLUSIVE OR OF A LIST OF INTEGERS
With no arguments returns identity 0
<expr> the integers
returns the result of the inclusive or operation

(logxor [<expr>…]) THE BITWISE EXCLUSIVE OR OF A LIST OF INTEGERS
With no arguments returns identity 0
<expr> the integers
returns the result of the exclusive or operation

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

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

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

(logtest <expr1> <expr2>) TEST BITWISE AND OF TWO INTEGERS

Defined in common.lsp when the bignum extension not loaded.

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

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

Part of the bignum extension.

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

(logcount <expr>) COUNT BITS IN AN INTEGER

Part of the bignum extension.

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

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

Part of the bignum extension.

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

(ash <expr1> <expr2>) ARITHMETIC SHIFT

Part of the math extension.

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

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

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

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

(ldb <spec> <int>) LOAD BYTE

Defined in common2.lsp. ldb can be used with setf, in which case it performs a stb followed by a setf into the field.

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

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

Defined in common2.lsp.

<spec>	specifier of byte to test
<int>	integer containing byte to test
returns	t if byte is zero, else nil

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

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

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

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

Defined in common2.lsp.

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

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

Defined in common2.lsp.

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

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
- returns t if the value is nil, nil otherwise

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

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

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

(constantp <expr>) IS THIS A CONSTANT?
- <expr> the expression to check
- returns t if the value is a constant (basically, would EVAL <expr> repeatedly return the same thing?), nil otherwise

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

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

(floatp <expr>) IS THIS A FLOAT?

<expr> the expression to check
returns t if the value is a float, nil otherwise

(rationalp <expr>) IS THIS A RATIONAL NUMBER?

Part of the bignum extension.
<expr> the expression to check
returns t if the value is rational (integer or ratio), nil otherwise

(realp <expr>) IS THIS A REAL NUMBER?

Defined in common2.lsp.
<expr> the expression to check
returns t if the value is rational or float, nil otherwise

(complexp <expr>) IS THIS A COMPLEX NUMBER?

Part of the math extension.
<expr> the expression to check
returns t if the value is a complex number, nil otherwise

(stringp <expr>) IS THIS A STRING?

<expr> the expression to check
returns t if the value is a string, nil otherwise

(characterp <expr>) IS THIS A CHARACTER?

<expr> the expression to check
returns t if the value is a character, nil otherwise

(arrayp <expr>) IS THIS AN ARRAY?

<expr> the expression to check
returns t if the value is an array, nil otherwise

(array-in-bounds-p <expr> <index>) IS ARRAY INDEX IN BOUNDS?

Defined in common2.lsp.
<expr> the array
<index> index to check
returns t if index is in bounds for the array, nil otherwise

(streamp <expr>) IS THIS A STREAM?

<expr> the expression to check
returns t if the value is a stream, nil otherwise

(open-stream-p <stream>) IS STREAM OPEN?

<stream> the stream
returns t if the stream is open, nil otherwise

(input-stream-p <stream>) IS STREAM READABLE?

<stream> the stream
returns t if stream is readable, nil otherwise

(output-stream-p <stream>) IS STREAM WRITABLE?

<stream>	the stream
returns	t if stream is writable, nil otherwise

(objectp <expr>) IS THIS AN OBJECT?

<expr>	the expression to check
returns	t if the value is an object, nil otherwise

(classp <expr>) IS THIS A CLASS OBJECT?

<expr>	the expression to check
returns	t if the value is a class object, nil otherwise

(hash-table-p <expr>) IS THIS A HASH TABLE?

Defined in common2.lsp.

<expr>	the expression to check
returns	t if the value is a hash table, nil otherwise

(keywordp <expr>) IS THIS A KEYWORD?

Defined in common2.lsp.

<expr>	the expression to check
returns	t if the value is a keyword symbol, nil otherwise

(packagep <expr>) IS THIS A PACKAGE?

Defined in common2.lsp.

<expr>	the expression to check
returns	t if the value is a package, nil otherwise

(boundp <sym>) IS A VALUE BOUND TO THIS SYMBOL?

<sym>	the symbol
returns	t if a value is bound to the symbol, nil otherwise

(fboundp <sym>) IS A FUNCTIONAL VALUE BOUND TO THIS SYMBOL?

<sym>	the symbol
returns	t if a functional value is bound to the symbol, nil otherwise

(functionp <sym>) IS THIS A FUNCTION?

Defined in common.lsp.

<expr>	the expression to check
returns	t if the value is a function -- that is, it can be applied to arguments. This is true for a closure, or subr. Otherwise returns nil.

(minusp <expr>) IS THIS NUMBER NEGATIVE?

<expr>	the number to test
returns	t if the number is negative, nil otherwise

(zerop <expr>) IS THIS NUMBER ZERO?

<expr>	the number to test
returns	t if the number is zero, nil otherwise

(plusp <expr>) IS THIS NUMBER POSITIVE?

<expr>	the number to test
returns	t if the number is positive, nil otherwise

(evenp <expr>) IS THIS INTEGER EVEN?

<expr>	the integer to test
returns	t if the integer is even, nil otherwise

(oddp <expr>) IS THIS INTEGER ODD?

<expr>	the integer to test
returns	t if the integer is odd, nil otherwise

(subsetp <list1> <list2> &key :test :test-not :key) IS SET A SUBSET?

<list1>	the first list
<list2>	the second list
:test	test function (defaults to eql)
:test-not	test function (sense inverted)
:key	function to apply to test function arguments (defaults to identity)
returns	t if every element of the first list is in the second list, nil otherwise

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

equalp defined in common.lsp.

<expr1>	the first expression
<expr2>	the second expression
returns	t if equal, nil otherwise. Each is progressively more liberal in what is "equal": eq: identical pointers -- works with characters, symbols, and arbitrarily small integers eql: works with all numbers, if same type (see also = on page 82) equal: lists and strings equalp: case insensitive characters (and strings), numbers of differing types, arrays (which can be equalp to string containing same elements)

(typep <expr> <type>) IS THIS A SPECIFIED TYPE?

<expr> the expression to test

<type> the type specifier. Symbols can either be one of those listed under type-of (on page 111) or one of:

ATOM	any atom
NULL	NIL
LIST	matches NIL or any cons cell
STREAM	any stream
NUMBER	any numeric type
REAL	flonum or rational number
INTEGER	fixnum or bignum
RATIONAL	fixnum or ratio
STRUCT	any structure (except hash-table)
FUNCTION	any function, as defined by functionp (page 88)

The specifier can also be a form (which can be nested). All form elements are quoted. Valid form CARs:

or	any of the cdr type specifiers must be true
and	all of the cdr type specifiers must be true
not	the single cdr type specifier must be false
satisfies	the result of applying the cdr predicate function to <expr>
member	<expr> must be eql to one of the cdr values
object	<expr> must be an object, of class specified by the single cdr value. The cdr value can be a symbol which must evaluate to a class.

Note: everything is of type T, and nothing is of type NIL

returns t if <expr> is of type <type>, nil otherwise

CONTROL CONSTRUCTS

(cond <pair>…) EVALUATE CONDITIONALLY

fsubr.

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

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

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

fsubr.

<expr> the expressions to be ANDed

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

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

fsubr.

<expr> the expressions to be ORed

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

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

fsubr.

<texpr> the test expression

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

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

returns the value of the selected expression

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

fsubr.

<texpr> the test expression

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

returns the value of the last expression or nil

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

fsubr.

<texpr> the test expression

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

returns the value of the last expression or nil

(case <expr> <case>…[(t <expr>)]) SELECT BY CASE

fsubr.

<expr> the selection expression

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

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

returns the value of the last expression of the matching case

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

Defined as macro in common2.lsp.

<expr> the selection expression

<case> pair consisting of:
(<type> <expr>…)
where:
<type> type specifier as in function TYPEP (page 90)
<expr> are expressions to execute if the case matches

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

returns the value of the last expression of the matching case

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

fsubr.

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

<expr> the expressions to be evaluated

returns the value of the last expression

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

fsubr.

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

<expr> the expressions to be evaluated

returns the value of the last expression

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

fsubr.

<sym> the catch tag

<expr> expressions to evaluate

returns the value of the last expression or the throw expression

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

fsubr.

<sym>	the catch tag
<expr>	the value for the catch to return (default is nil)
returns	never returns

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

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

fsubr.

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

LOOPING CONSTRUCTS

(loop <expr>…) BASIC LOOPING FORM

fsubr.

<expr> the body of the loop

returns never returns (must use non-local exit, such as RETURN)

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

fsubr. do binds simultaneously, do* binds sequentially

<binding> the variable bindings each of which is either:

1) a symbol (which is initialized to nil)

2) a list of the form: (<sym> <init> [<step>])

where:

<sym> is the symbol to bind

<init> the initial value of the symbol

<step> a step expression

<texpr> the termination test expression

<rexpr> result expressions (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 (default is nil)

<expr> the body of the loop (treated like an implicit prog)

returns the result expression

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

fsubr.

<sym> the symbol to bind to each value from 0 to n-1

<expr> the number of times to loop (a fixnum)

<rexpr> the result expression (default is nil)

<expr> the body of the loop (treated like an implicit prog)

returns the result expression

THE PROGRAM FEATURE

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

fsubr -- equivalent to (let () (block nil (tagbody …)))

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

<expr> expressions to evaluate or tags (symbols)

returns nil or the argument passed to the return function

(block <name> <expr>…) NAMED BLOCK

fsubr.

<name> the block name (quoted symbol)

<expr> the block body

returns the value of the last expression

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

fsubr.

<expr> the value (default is nil)

returns never returns

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

fsubr. In traditional XLISP, the names are dynamically scoped. A compilation option (default) uses lexical scoping like Common Lisp.

<name> the block or function name (quoted symbol). If name is nil, use function RETURN.

<value> the value to return (default is nil)

returns never returns

(tagbody <expr>…) BLOCK WITH LABELS

fsubr.

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

returns nil

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

fsubr. In traditional XLISP, tags are dynamically scoped. A compilation option (default) uses lexical scoping like Common Lisp.

<sym> the tag (quoted)

returns never returns

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

fsubr.

<slist> list of symbols (evaluated)

<vlist> list of values to bind to the symbols (evaluated)

<expr> expression(s) to evaluate

returns the value of the last expression

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

fsubr.

<expr1> the first expression to evaluate

<expr> the remaining expressions to evaluate

returns the value of the first expression

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

fsubr.

<expr1> the first expression to evaluate

<expr2> the second expression to evaluate

<expr> the remaining expressions to evaluate

returns the value of the second expression

(progn <expr>…) EXECUTE EXPRESSIONS SEQUENTIALLY

fsubr.

<expr> the expressions to evaluate

returns the value of the last expression (or nil)

INPUT/OUTPUT FUNCTIONS

Note that when printing objects, printing is accomplished by sending the message :prin1 to the object.

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

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

<stream>	the input stream (default, or nil, is *standard-input*, t is *terminal-io*)
<eofp>	When t, signal an error on end of file, when nil return <eof> (default is t)
<eof>	the value to return on end of file (default is nil)
<rflag>	recursive read flag. The value is ignored
returns	the expression read

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

Defined in init.lsp

<ch>	character to define
<fcn>	function to bind to character (see page 30)
returns	t if TMACRO rather than NMACRO

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

Defined in init.lsp.

<ch>	character
returns	function bound to character

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

The expression is printed using prin1, then current line is terminated. Note: this is backwards from Common Lisp.

<expr>	the expression to be printed
<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	the expression

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

symbols, cons cells (without circularities), arrays, strings, numbers, and characters are printed in a format generally acceptable to the read function. Printing format can be affected by the global formatting variables: *print-level* and *print-length* for lists and arrays, *print-base* for rationals, *integer-format* for fixnums, *float-format* for flonums, *ratio-format* for ratios, and *print-case* and *readtable-case* for symbols.

<expr>	the expression to be printed
<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	the expression

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

Like PRIN1 except symbols (including uninterned), strings, and characters are printed without using any quoting mechanisms.

<expr>	the expressions to be printed
<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	the expression

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

Uses prin1 for printing.

<expr>	the expressions to be printed
<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	the expression

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

<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	nil

(fresh-line [<stream>]) START A NEW LINE

<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	t if a new list was started, nil if already at the start of a line

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

<expr>	the expression
returns	the length

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

<expr>	the expression
returns	the length

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

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

<fmt>	optional format string for question (see page 99)
<arg>	arguments, if any, for format string
returns	t for yes, nil for no

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

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

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

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

Defined in common2.lsp.

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

THE FORMAT FUNCTION

(format <stream> <fmt> [<arg>…]) DO FORMATTED OUTPUT

<stream>	the output stream (t is *standard-output*)
<fmt>	the format string
<arg>	the format arguments
returns	output string if <stream> is nil, nil otherwise

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

~?	use next argument as recursive format string
~(~)	process format string with case conversion
~{ ~}	process format string repetitively
~^	escape upwards
~*	skip arguments
~%	start a new line
~&	start a new line if not on a new line
~\n	ignore return and following whitespace
~|	start a new page
~~	print a tilde character
~A or ~a	print next argument using princ
~B or ~b	print next argument as binary integer (bignum extension)
~D or ~d	print next argument as decimal integer
~E or ~e	print next argument in exponential form
~F or ~f	print next argument in fixed point form
~G or ~g	print next argument using either ~E or ~F depending on magnitude
~O or ~o	print next argument as octal integer
~P or ~p	print plurals
~R or ~r	print next number in any radix, in English or Roman numerals (bignum extension)
~S or ~s	print next argument using prin1
~T or ~t	go to a specified column
~X or ~x	print next argument as hexadecimal integer
~[~; ~]	process format string conditionally

(Note that ~R for English and Roman numeral output, ~P, ~^, comma insertion in ~B, ~D, ~O, ~R and ~X, and some functionality of ~* are not present in all compiled copies of XLISP. In particular these may only be in the Qt compiled versions.)

The format directives can contain optional prefix and optional colon (:) or at-sign (@) modifiers between the tilde and directive character. Prefix characters are signed integers, the character '#' which represents the remaining number of arguments, the character 'v' to indicate the number is taken from the next argument, or a single quote (') followed by a single character for those parameters that should be a single character. Note that except for the ~^ directive, integer values must be in the range of 0 to 255. This prevents potentially disasterous results like million character long print lines.

For ~A and ~S the full form is:

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

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

~15,,2,'.@A

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

For ~D, ~B, ~O, and ~X the full form is ("D" shown):

~mincol,padchar,cchar,ccount:@D

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

If : is used, cchar (default is a comma) is inserted ever ccount (default 3) digits from the right.

For ~R, the full form is:

~radix,mincol,padchar,cchar,ccount:@R

The radix must be in the range 2 through 36. Other arguments are as in ~D, above. If no argument values are given, ~R prints a (fixnum) value as an English cardinal number, ~:R prints as a English ordinal number, ~@R prints a Roman numeral value, and ~:@R prints an old style (example: *IIII* instead of *IV*) Roman numeral value.

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

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

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

For ~P if the next argument is not eql 1 an “s” is printed. If the @ modifier is used, “ies” is printed if the next argument is not eql 1 and “y” is printed if eql 1. The : modifier causes the previous argument to be used rather than a new argument.

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

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

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

For ~(, the full form is ~(string~). The string is processed as a format string, however case conversion is performed on the output. If no modifiers are used, the string is converted to lowercase. If the colon

modifier is used alone then all words are capitalized. If the @ modifier is used alone then the first character is converted to upper case and all remaining to lowercase. If both modifiers are used, all characters are converted to uppercase.

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

The ~^ directive escapes (exits) the nearest enclosing ~{, ~(, or ~? directive. For the ~{ directive the escape is of the current iteration. If not in any of these, it will finish the format command. The escape is conditional. With no prefix modifiers the escape occurs if there are no remaining arguments. Only within a ~:{ the ~:^ directive can be used to escape if there are no remaining sublists. When there are prefix modifiers, a single modifier must be 0 for the escape to occur, two modifiers must be equal, and if there are three modifiers then the second must be greater than or equal to the first and the third must be greater than or equal to the second. The modifiers can be any fixnum value.

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

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

For ~T, the full form is:

~count,tabwidth@T

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

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

FILE I/O FUNCTIONS

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

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

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

TERMINAL-IO should not be changed. Any other system streams that are changed by an application should be restored to their original values.

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

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

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

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

<flag>	flag for skipping white space (default is nil)
<stream>	the input stream (default, or nil, is *standard-input*, t is *terminal-io*)
<eofp>	When t, signal an error on end of file, when nil return <eof> (default is t)
<eof>	the value to return on end of file (default is nil)
returns	the character or <eof> at end of file

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

<ch>	the character to write
<stream>	the output stream (default, or nil, is *standard-output*, t is *terminal-io*)
returns	the character

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

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

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

(open <fname> &key :direction :element-type :if-exists :if-does-not-exist) OPEN A FILE STREAM

Note: A maximum of ten files can be open at any one time, including any files open via the LOAD, DRIBBLE, SAVE and RESTORE commands. The open command may force a garbage collection to reclaim file slots used by unbound file streams.

<fname>	the file name string, symbol, or file stream created via OPEN. In the last case, the name is used to open a second stream on the same file -- this can cause problems if one or more streams is used for writing.
:direction	Read and write permission for stream (default is :input)
:input	Open file for read operations only.
:prob	Open file for reading, then close it (use to test for file existence)
:output	Open file for write operations only.
:io	Like :output, but reading also allowed.
:element-type	FIXNUM or CHARACTER (default is CHARACTER), as returned by type-of function (on page 111), or UNSIGNED-BYTE, SIGNED-BYTE, (UNSIGNED-BYTE <size>), or (SIGNED-BYTE <size>) with the bignum extension. CHARACTER (the default) is for text files, the other types are for binary files and can only be used with READ-BYTE and WRITE-BYTE. FIXNUM is a vestige of older XLISP-PLUS releases and is identical to (UNSIGNED-BYTE 8). If no size is given, then size defaults to 8. Size must be a multiple of 8.
:if-exists	action to take if file exists. Argument ignored for :input (file is positioned at start) or :probe (file is closed)
:error	give error message
:rename	rename file to generated backup name, then open a new file of the original name. This is the default action
:new-version	same as :rename
:overwrite	file is positioned to start, original data intact
:append	file is positioned to end
:supersede	delete original file and open new file of the same name
:rename-and-delete	same as :supersede
nil	close file and return nil
:if-does-not-exist	action to take if file does not exist.
:error	give error message (default for :input, or :overwrite or :append)
:create	create a new file (default for :output or :io when not :overwrite or :append)
nil	return nil (default for :probe)
returns	a file stream, or sometimes nil

(close <stream>) CLOSE A FILE STREAM

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

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

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

Defined in common2.lsp.

<fname>	file name string or symbol
returns	t if file exists, else nil

(delete-file <fname>) DELETE A FILE

<fname> file name string, symbol or a stream opened with OPEN

returns t if file does not exist or is deleted. If <fname> is a stream, the stream is closed before the file is deleted. An error occurs if the file cannot be deleted.

(truename <fname>) OBTAIN THE FILE PATH NAME

<fname> file name string, symbol, or a stream opened with OPEN

returns string representing the true file name (absolute path to file)

(with-open-file (<var> <fname> [<karg>…]) [<expr>…]) EVALUATE USING A FILE

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

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

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

<fname> file name string or symbol

<stream> a file or string stream

<karg> keyword arguments for the implicit open command

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

returns value of last <expr>

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

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

<stream> the input stream

<eofp> When t, signal an error on end of file, when nil return <eof> (default is t)

<eof> the value to return on end of file (default is nil)

returns the byte (integer) or <eof> at end of file

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

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

<byte> the byte to write (integer)

<stream> the output stream

returns the byte (integer)

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

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

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

returns length of file, or nil if cannot be determined

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

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

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

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

returns if setting position, and successful, then t; if getting position and successful then the position; otherwise nil

STRING STREAM FUNCTIONS

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

An unnamed input stream is set-up 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>]]) CONVERT STRING TO STREAM

<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) ACCUMULATE OUTPUT IN A STREAM

returns	an unnamed output stream

(get-output-stream-string <stream>) CONVERT STREAM TO STRING

The output stream is emptied by this function

<stream>	the output stream
returns	the output so far as a string

(get-output-stream-list <stream>) CONVERT STREAM TO LIST

The output stream is emptied by this function

<stream>	the output stream
returns	the output so far as a list

(with-input-from-string (<var> <str> &key :start :end :index) [<expr>…]) EVALUATE READING FROM A STRING

Defined as macro in common.lsp.

<var>	symbol that stream is bound to during execution of expressions (quoted)
<str>	the string
:start	starting offset into string (default is 0)
:end	ending offset + 1 (default, or nil, is end of string)
:index	setf place form which gets final index into string after last expression is executed (quoted)
<expr>	expressions to evaluate (implicit progn)
returns	the value of the last <expr>

(with-output-to-string (<var>) [<expr>…]) EVALUATE WRITING TO A STRING

Defined as macro in common.lsp.

<var>	symbol that stream is bound to during execution of expressions (quoted)
<expr>	expressions to evaluate (implicit progn)
returns	contents of stream, as a string

DEBUGGING AND ERROR HANDLING FUNCTIONS

Apart from these debugging functions, there are some extra utilities distributed with this package. See the section starting on page 117.

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

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

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

fsubr. If no functions given, all functions are removed from the trace list.
<sym> the function(s) to remove (quoted)
returns the trace list

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

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

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

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

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

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

(clean-up) CLEAN-UP AFTER AN ERROR

returns never returns

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

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

(continue) CONTINUE FROM A CORRECTABLE ERROR

returns never returns

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

fsubr.
<expr> the expression to execute
<pflag> flag to control printing of the error message (default is t)
returns the value of the last expression consed with nil or nil on error

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

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

returns nil

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

<expr> the expression to evaluate. <ehook> is not used at the top level.

<ehook> the value for *evalhook*

<ahook> the value for *applyhook*

<env> the environment (default is nil). The format is a dotted pair of value (car) and function (cdr) binding lists. Each binding list is a list of level binding a-lists, with the innermost a-list first. The level binding a-list associates the bound symbol with its value.

returns the result of evaluating the expression

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

<fun> The function closure. <ahook> is not used for this function application.

<arglist> The list of arguments

<ehook> the value for *evalhook*

<ahook> the value for *applyhook*

returns the result of applying <fun> to <arglist>

(debug) ENABLE DEBUG BREAKS

(nodebug) DISABLE DEBUG BREAKS

Defined in init.lsp

returns debug returns t
nodebug returns nil

(ecase <expr> <case>…) SELECT BY CASE

(ccase <expr> <case>…)

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

<expr> the selection expression

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

returns the value of the last expression of the matching case

(etypecase <expr> <case>…) SELECT BY TYPE

(ctypecase <expr> <case>…)

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

<expr> the selection expression

<case> pair consisting of:
(<type> <expr>…)
where:
<type> type specifier as in function TYPEP (page 90)
<expr> are expressions to execute if the case matches

returns the value of the last expression of the matching case

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

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

<place>	a valid field specifier (generalized variable)
<type>	a valid type specifier as in function TYPEP (page 90)
<string>	string to print as the error message
returns	nil

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

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

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

(the <type> <form>) DECLARE TYPE OF A FORM

Defined as macro in common.lsp, and provided to assist in porting Common Lisp applications to XLISP-PLUS. If type of <form> is not <type> then an error is signaled.

<type>	the declared type (quoted)
<form>	the form to evaluate
returns	the value of form

SYSTEM FUNCTIONS

(load <fname> &key :verbose :print) LOAD A SOURCE FILE

An implicit ERRSET exists in this function so that if error occurs during loading, and *breakenable* is nil, then the error message will be printed and nil will be returned. The OS environmental variable XLPATH is used as a search path for files in this function. If the filename does not contain path separators ('/' for UNIX, and either '/' or '\' for MS-DOS) and XLPATH is defined, then each pathname in XLPATH is tried in turn until a matching file is found. If no file is found, then one last attempt is made in the current directory. The pathnames are separated by either a space or semicolon, and a trailing path separator character is optional.

<fname>	the filename string, symbol, or a file stream created with OPEN. The extension "lsp" is assumed.
:verbose	the verbose flag (default is t)
:print	the print flag (default is nil)
returns	t if successful, else nil

(restore <fname>) RESTORE WORKSPACE FROM A FILE

The OS environmental variable XLPATH is used as a search path for files in this function. See the note under function "load", above. The standard system streams are restored to the defaults as of when XLISP-PLUS was started. Files streams are restored in the same mode they were created, if possible, and are positioned where they were at the time of the save. If the files have been altered or moved since the time of the save, the restore will not be completely successful. Memory allocation will not be the same as the current settings of ALLOC are used. Execution proceeds at the top-level read-eval-print loop. The state of the transcript logging is not affected by this function.

<fname>	the filename string, symbol, or a file stream created with OPEN. The extension "wks" is assumed.
returns	nil on failure, otherwise never returns

(save <fname>) SAVE WORKSPACE TO A FILE

You cannot save from within a load. Not all of the state may be saved -- see "restore", above. By saving a workspace with the name "xlisp", that workspace will be loaded automatically when you invoke XLISP-PLUS.

<fname>	the filename string, symbol, or a file stream created with OPEN. The extension "wks" is assumed.
returns	t if workspace was written, nil otherwise

(savefun <fcn>) SAVE FUNCTION TO A FILE

Defined in init.lsp

<fcn>	function name (saves it to file of same name, with extension ".lsp")
returns	t if successful

(dribble [<fname>]) CREATE A FILE WITH A TRANSCRIPT OF A SESSION

<fname>	file name string, symbol, or file stream created with OPEN. If missing, close current transcript
returns	t if the transcript is opened, nil if it is closed

(gc) FORCE GARBAGE COLLECTION

returns	nil

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

<num> the (fixnum) number of segments to add (default is 1)

returns the (fixnum) number of segments added

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

<num> the (fixnum) number of nodes to allocate

<num2> the (fixnum) number of pointer elements to allocate in an array segment (when dynamic array allocation compiled). Default is no change.

<num3> the <fixnum> ideal ratio of free to used vector space (versions of XLISP using dldmem.c). Default is 1. Increase if extensive time is spent in garbage collection in bignum math intensive programs.

returns the old number of nodes to allocate

(room) SHOW MEMORY ALLOCATION STATISTICS

Statistics (which are sent to *STANDARD-OUTPUT*) include:

Nodes - number of nodes, free and used

Free nodes - number of free nodes

Segments - number of node segments, including those reserved for characters and small integers.

Allocate - number of nodes to allocate in any new node segments

Total - total memory bytes allocated for node segments, arrays, and strings

Collections - number of garbage collections

Time - time spent performing garbage collections (in seconds)

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

Vector nodes - total vector space (pointers and string, in pointer sized units)

Vector free - free space in vector area (may be fragmented across segments)

Vector segs - number of vector segments. Increases and decreases as needed.

Vec allocate - number of pointer elements to allocate in any new vector segment

Vec collect - number of garbage collections instigated by vector space exhaustion

returns nil

(time <expr>) MEASURE EXECUTION TIME

fsubr.

<expr> the expression to evaluate

returns the result of the expression. The execution time is printed to *TRACE-OUTPUT*

(sleep <expr>) TIME DELAY

Defined in common2.lsp.

<expr> time in seconds

returns nil, after <expr> seconds delay

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

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

returns integer time in system units (see internal-time-units-per-second on page 39). meaning of absolute values is system dependent

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

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

<expr>	the expression to coerce
<type>	desired type, as returned by type-of (see page 111)
returns	<expr> if type is correct, or converted object

(type-of <expr>) RETURNS THE TYPE OF THE EXPRESSION

It is recommended that typep be used instead, as it is more general. In the original XLISP, the value nil was returned for NIL.

<expr>	the expression to return the type of	
returns	One of the symbols:	
	LIST	for NIL (lists, conses return CONS)
	SYMBOL	for symbols
	OBJECT	for objects
	CONS	for conses
	SUBR	for built-in functions
	FSUBR	for special forms
	CLOSURE	for defined functions
	STRING	for strings
	FIXNUM	for integers
	BIGNUM	for large integers
	RATIO	for ratios
	FLONUM	for floating point numbers
	COMPLEX	for complex numbers
	CHARACTER	for characters
	FILE-STREAM	for file pointers
	UNNAMED-STREAM	for unnamed streams
	ARRAY	for arrays
	HASH-TABLE	for hash tables
	sym	for structures of type "sym"

(peek <addrs>) PEEK AT A LOCATION IN MEMORY

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

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

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

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

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

(get-key) READ A KEYSTROKE FROM CONSOLE

OS dependent.

returns	integer value of key (no echo)

(system <command>) EXECUTE A SYSTEM COMMAND

Note: OS dependent -- not always available.

<command> Command string, if 0 length then spawn OS shell

returns t if successful. Note: MS-DOS command.com always returns success.

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

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

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

returns the previous value

(top-level-loop) DEFAULT TOP LEVEL LOOP

Runs the XLISP top level read-eval-print loop, described earlier.

returns never returns

(reset-system) FLUSH INPUT BUFFERS

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

returns nil

(exit) EXIT XLISP

returns never returns

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

<expr> the expression to copy

returns nil if value is nil and NILSYMBOL compilation option not declared, otherwise if type is:

SYMBOL	copy as an ARRAY
OBJECT	copy as an ARRAY
CONS	(CONS (CAR <expr>)(CDR <expr>))
CLOSURE	copy as an ARRAY
STRING	copy of the string
FIXNUM	value
FLONUM	value
RATIO	value
CHARACTER	value
UNNAMED-STREAM	copy as a CONS
ARRAY	copy of the array
COMPLEX	copy as an ARRAY
HASH-TABLE	copy as an ARRAY
BIGNUM	copy as a string
structure	copy as an ARRAY

(eval-when <condition> <body> …) EVALUATE ONLY DURING LOAD OR EXECUTION TIME

Defined as macro in common.lsp, and provided to assist in porting Common Lisp applications to XLISP-PLUS.

<condition> List of conditions

<body> expressions which are evaluated if one of the conditions is EXECUTE or LOAD

returns result of last body expression

GRAPHICS FUNCTIONS

The following graphic and display functions represent an extension by Tom Almy. There are three versions of graphics, the original MS-DOS, Windows, and Qt. In the MS-DOS and Windows version all text and graphics appear in a single window while in the Qt version there is a separate graphics window that can have text written to it.

Qt has two modes for display – Qt-graphic and Qt-text. Text mode is entered with the command prompt, error or break messages, or executing Mode 0, 1, 2, or 3, or gmode nil. Graphics mode is entered with the mode command > 3 or any move or draw function. If the graphics window is closed, commands which enter graphic mode will make the window visible. All graphics output is buffered for performance. The display is updated whenever text mode is entered or a mode or gmode command is executed. The file *qt.lsp* defines extra functions to make the graphics package easier to use. The extra functions are in the GRAPHIC package.

(cls) CLEAR DISPLAY

Clear the display and position cursor at upper left corner. Qt- if in graphic mode, clears the graphic display while if in text mode clears the text and graphics display.

returns nil

(cleol) CLEAR TO END OF LINE

Clears current line to end. In Qt this is text mode only.

returns nil

(goto-xy [<column> <row>]) GET OR SET CURSOR POSITION

Cursor is repositioned if optional arguments are specified. Coordinates are clipped to actual size of display. In Qt graphics mode will do a MOVE operation expecting a text write to the display to follow.

<column> 0-based column (x coordinate)

<row> 0-based row (y coordinate)

returns list of original column and row positions

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

Standard modes require only <ax> argument. Extended modes are "Super-VGA" or "Super-EGA" and are display card specific. Not all XLISP versions support all modes. For Windows this is a no-op except for the return values. For Qt, values of 0-3 set text mode, others in the table set an equivalent screen size, while unlisted values with width and height specified set a custom size. Applications not requiring compatibility should use the GMODE function in qt.lsp.

<ax> Graphic mode (value passed in register AX)

Common standard Modes:

0,1 - 40x25 text

2,3 - 80x25 text

4,5 - 320x200 4 color graphics (CGA)

6 - 640x200 monochrome graphics (CGA)

13 - 320x200 16 color graphics (EGA)

14 - 640x200 16 color graphics (EGA)

16 - 640x350 16 color graphics (EGA)

18 - 640x480 16 color graphics (VGA)

19 - 320x200 256 color graphics (VGA)

<bx> BX value for some extended graphic modes

<width> width for extended graphic modes
<height> height for extended graphic modes
returns a list of the number of columns, number of lines (1 for CGA), maximum X graphic coordinate (-1 for text modes), and the maximum Y graphic coordinate (-1 for text modes), or nil if fails

(color <value>) SET DRAWING COLOR
(color <fr> <fg> <fb> [
 <bg> <bb>])

The first version sets a 256 color palate value where the 3 least significant bits are RGB foreground with bit 3 adding intensity, bits 4-6 are RBG background, bit 7 is exclusive OR (doesn't work in Qt) and bit 8 is background intensity. The second version, available in Windows and Qt, sets 256 levels for red, green, and blue, and optionally for the background as well. Add 256 to the red foreground in Windows for exclusive OR.

<value> Drawing color (not checked for validity)
<fr> <fg> <fb> Foreground RGB colors (not checked for validity)

 <bg> <bb> Background RGB colors. Unchanged if not specified.
returns arguments

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

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

<x1> <y1> Point(s) drawn to, in order
returns t if succeeds, else nil

(font <size> <family> <style>) SELECT GRAPHICS FONT

Part of Qt graphics. Graphics must first be initialized with MODE or GMODE function.

<size> size in points
<family> 1 – Helvetica, 2 – Times, 0 or other FIXNUM – Courier. These can also be specified by constants defined in qt.lsp graphic: helvetica times and courier.
<style> 0 – Normal, 1 – Bold, 2 – Italic, 3 – Bold and Italic. These can also be specified by constants defined in qt.lsp graphic: normal bold italic and bold-italic.
returns T

(brush <color> <style>) DEFINE FILL BRUSH

Part of Qt graphics. Graphics must first be initialized with MODE or GMODE function.

<color> Either a single fixnum color or a list of three colors, Red, Green, and Blue. Color values can be specified by constants defined in qt.lsp graphic:black blue green cyan red magenta brown ltgrey grey ltblue ltgreen ltcyan ltred ltmagenta yellow or white.
<style> 0 – no fill, 1 – solid fill, 2-8 heavy through light shade, 9 – horizontal lines, 10 – vertical lines, 11 – cross lines, 12 - / diagonal slashes, 13 - \ diagonal slashes, 14 – diagonal cross pattern. These are also defined by constants in qt.lsp, graphic:nofill, solid, dense1, dense2, dense3, dense4, dense5, dense6, dense7, horlines, verlines, crosslines, slash, backslash, and diagcross.

returns T

(drawrect <x1> <y1> <width> <height>) DRAW A RECTANGLE

Part of Qt graphics. Outline drawn in COLOR and filled in BRUSH.

<x1><y2> Coordinates of one corner

<width> Width of rectangle (can be negative)

<height> Height of rectangle (can be negative)

returns T

(drawellipse <x> <y> <width> [<height>]) DRAW AN ELLIPSE

Part of Qt graphics. Outline drawn in COLOR and filled in BRUSH.

<x><y> Coordinates of corner

<width> Width of ellipse. Can be negative!

<height> Height of ellipse. If not given, same as width (a circle). Can be negative!

returns T

(graphic:gmode <x> [<y>]) CONTROL GRAPHICS MODE

Defined in qt.lsp for Qt graphics. This is an alternative to the MODE function. Applications should not mix the use of MODE and GMODE. This function sets the variables graphic:*columns* *rows* *xsize* *ysize* *charwidth* and *charheight* as a side effect when first argument is not T or NIL.

<x> If NIL sets text mode and updates the display. If T sets graphics mode and updates display. If a fixnum, sets the desired width in pixels of the graphics display.

<y> Fixnum desired height of graphics display in pixels.

returns NIL if <x> is T or NIL otherwise returns a list of the number of columns, rows, width, and height.

(graphic:colors <foreground> [<background>]) SET THE FORE/BACK COLORS

(graphic:colors < fr> <fb> <fg> [
 <bg> <bb>])

Defined in qt.lsp for Qt graphics. This is an alterative to the COLOR function. Applications should not mix the use of COLOR and COLORS. This function sets the variables graphic:*forecolor* and *backcolor* to lists of RGB values for the foreground and background colors as a side effect.

<foreground> Either a list of RGB values in the range 0 to 255 or a fixnum color value (same as the COLOR function). Color values can be specified by constants graphic:black blue green cyan red magenta brown ltgrey grey ltblue ltgreen ltcyan ltred ltmagenta yellow or white.

<background> If specified, sets the background color as well. Either a list of R G B values or a fixnum color value (same as the COLOR function). Color values can be specified by constants graphic:black blue green cyan red magenta brown ltgrey grey ltblue ltgreen ltcyan ltred ltmagenta yellow or white.

<fr> <fg> <fb> Foreground RGB colors (not checked for validity)

 <bg> <bb> Background RGB colors. Unchanged if not specified.

returns a list of a list of the RGB foreground colors and a list of the RGB background

(graphic:drawrectcorners <x1> <y1> <x2> <y2>) DRAW A RECTANGLE

Defined in qt.lsp for Qt graphics. Same function as DRAWRECTANGLE however two corner coordinates are specified instead of center, height, and width.

<x1> <y1> X and Y coordinates of first corner

<x2> <y2> X and Y coordinates of opposing corner

returns T

(graphic:drawellipsecenter <x> <y> <w> [<h>]) DRAW AN ELLIPSE

Defined in qt.lsp for Qt graphics. Same function as DRAWELLIPSE however center coordinate is specified instead of a corner.

<x><y> coordinates of center
<w> width
<h> height (default is width, which gives a circle)
returns T

(graphic:point <x> <y>) DRAW A POINT

Defined in qt.lsp for Qt graphics. Draws a point, equivalent to (move x y x y).

<x> x coordinate of point
<y> y coordinate of point
returns T

ADDITIONAL FUNCTIONS AND UTILITIES

STEP.LSP

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

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

To invoke:

```
(step (whatever-form with args))
```

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

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

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

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

Global variables - newline, *hooklevel*

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

STEPPER.LSP

This is a more powerful stepper/debugger than step.lsp. The package is in stepper.lsp

To invoke the stepper:

(step <form with args>)

The stepper will stop and print every form, then wait for user input. Forms are printed compressed, i.e. to a depth and length of 3. This may be changed by assigning the desired depth and length values to *pdepth* and *plen* before invoking the stepper, or from within the stepper via the . and # commands.

For example, suppose you have the following defined:

```
(defun fib (n)
  (if (or (= n 1) (= n 2))
      1
      (+ (fib (1- n)) (fib (- n 2)))))
```

Then (step (fib 4)) will produce the following:

```
0 >==> (fib 4)
 1 >==> (if (or (= n 1) (= n 2)) 1 ...) :
```

The colon is the stepper's prompt.

For a list of commands, type h. This produces the following:

```
Stepper Commands
n or space        next form
s or <cr>         step over form
f FUNCTION        go until FUNCTION is called
b FUNCTION        set breakpoint at FUNCTION
b <list>          set breakpoint at each function in list
c FUNCTION        clear breakpoint at FUNCTION
c <list>          clear breakpoint at each function in list
c *all*           clear all breakpoints
g                 go until a breakpoint is reached
u                 go up; continue until enclosing form is done
w                 where am I? -- backtrace
t                 toggle trace on/off
q                 quit stepper, continue execution
p                 pretty-print current form (uncompressed)
e                 print environment
x <expr>          execute expression in current environment
r <expr>          execute and return expression
# nn              set print depth to nn
. nn              set print length to nn
h                 print this summary
```

- Breakpoints may be set with the b command. You may set breakpoints at one function, e.g. b FOO<cr> sets a breakpoint at the function FOO, or at various functions at once, e.g. b (FOO FIE FUM)<cr> sets breakpoints at the functions FOO, FIE, and FUM. Breakpoints are cleared with the c command in an analogous way. Furthermore, a special form of the c command, c *all* <cr>, clears all previously set breakpoints. Breakpoints are remembered from one invocation of step to the next, so it is only necessary to set them once in a debugging session.
- The g command causes execution to proceed until a breakpoint is reached, at which time more stepper commands can be entered.

- The f command sets a temporary breakpoint at one function, and causes execution to proceed until that function is called.
- The u command continues execution until the form enclosing the current form is done, then re-enters the stepper.
- The w command prints a back trace.
- The q command quits and causes execution to continue uninterrupted.
- Entry and exit to functions are traced after a g, f, u, or q command. To turn off tracing, use the t command which toggles the trace on/off.
- Also, with trace off, the values of function parameters are not printed.
- The s command causes the current form to be evaluated.
- The n command steps into the current form.
- The . and # commands change the compression of displayed forms. E.g. in the previous example:

```
1 >==> (if (or (= n 1) (= n 2)) 1 ...) : . 2
1 >==> (if (or (= n 1) (= n 2)) ...) ...) ...) :
```

changes the print length to 2, and

```
1 >==> (if (or (= n ...) ...) ...) : # 2
1 >==> (if (or #\# ...) ...) :
```

changes the print depth to 2.

- To print the entire form use the p command, which pretty-prints the entire form.
- The e command causes the current environment to be printed.
- The x command causes an expression to be executed in the current environment. Note that this permits the user to alter values while the program is running, and may affect execution of the program.
- The r command causes the value of the given expression to be returned, i.e. makes it the return value of the current form.

PP.LSP

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

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

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

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

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

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

See the source file for more information.

DOCUMENT.LSP

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

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

Use with SETF to alter documentation string.

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

(glos <symbol> [t]) GET DOCUMENTATION
(glos <class> <sel>)

Defined in package TOOLS.

<symbol>	Either the symbol for which the documentation is requested, or a string which will match all symbol names containing that string
t	Flag saying to treat symbol as a string, and match all related names
<class>	Quoted class name (a sybol)
<sel>	Message selector. Gives documentation for the selector for the class.
returns	nothing

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

INSPECT.LSP

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

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

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

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

(describe <expr>) DESCRIBE

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

<expr>	Expression to describe
returns	The expression

The editor alters the current selection by copying so that aborting all changes is generally possible; the exception is when editing a closure, if the closure is backed out of, the change is permanent. Also, naturally, changing the values of structure elements, instance variables, or symbols cannot be undone.

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

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

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

(command list on next page)

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

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

ADDITIONAL COMMANDS (selection is a list or array):

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

MEMO.LSP

This file contains the utility memoize, which transforms a given function so that it will save the results of previous computations for later use. Whenever a memoized function is called, it will first look in its allocated memoization table to see if the result has been previously calculated and, if found, returns it immediately. Otherwise, it proceeds with the normal execution and, at the end, saves the result for future use. This utility can dramatically increase execution speed, specially with recursive functions. When the package feature is compiled in, this utility is in the TOOLS package.

As an example, let's use memoize with the function fib, which calculates the n-th Fibonacci number, as in the stepper section:

First, let's define the function fib:

```
>(defun fib (n)
  (if (or (= n 1) (= n 2))
      1
      (+ (fib (1- n)) (fibo (- n 2))) ))
```

Now let's see how long it takes to calculate Fib (10):

```
> (time (fib 20))
The evaluation took 9.40 seconds.
10946
```

Now let's use the memoization:

```
> (memoize 'fib)
#<Closure: #37a7034b>
> (time (fib 20))
The evaluation took 0.03 seconds.
10946
```

Quite a dramatic improvement!.

By default, a memoized function checks only its first argument to see if the result has been previously saved. If you want the memoized function to check more than one argument, use the optional :key parameter to compare with the desired number of parameters, and :test to make the comparison.

```
> (memoize ‘function-with-2-args :key #’(lambda (lst) (cons (first lst)(second lst))) :test #’equal)
```

To reset the memoizing table of a function, call clear-memoize:

```
> (clear-memoize ‘fib)
#<Hash-table: #2ebf38b2>
```

Calling unmemoize resets a given function to its previous definition.

```
> (unmemoize ‘fib)
```

PROFILE.LSP

This file contains a series of macros for a utility called profiling, which helps you monitor the execution of functions. It can be very useful when deciding what to optimize in a program. Profiling a function means monitoring how many times the function is called and how long does its execution take. After running a program containing profiled functions, a profile report can be printed with statistics comparing the use of those functions. Therefore, we know which part of the code is more sensitive to optimization. When the package feature is compiled in, this utility is in the TOOLS package.

Example:

```
> (profile foo fee fii)
(foo fee fii)
> (function-calling-foo-fee-fii)
...
> (profile-report)
Total elapsed time: 0.16 seconds.
  Count     Secs    Time%   Name
     10     0.00       0%   foo
      5     0.01       3%   fee
      5     0.30      97%   fii
```

Calling the function profile without arguments, returns the list of profiled functions:

```
> (profile)
(foo fee fii)
```

You can deselect one or more functions with unprofile:

```
> (unprofile foo fee)
```

There is provided a further macro called with-profiling, which encompasses the above process of profiling/executing/reporting/unprofiling in one single block. The following has the same effect as the previous steps input sequentially:

```
> (with-profiling (foo fee fii) (function-calling-foo-fee-fii))
```

LIVING WITH PACKAGES

Before Packages

In XLISP-PLUS (as in most traditional LISP implementations) symbols that are read in are looked up in an object array (or list) called *OBARRAY*. This is a hash table of all interned symbols. Because there is only one object array, every symbol name has a unique symbol associated with it.

Unfortunately, a problem occurs when it is desired to load more than one application into the XLISP-PLUS workspace. If two applications use the same symbol name for two different functions, then the first application will not run after the second is loaded. Likewise there can be problems if the symbols are used as variables or constants.

The Package Concept

Common Lisp solves this problem by providing "packages." A package is basically an independent object table that can be associated with individual projects. With separate object tables, symbol usage conflicts can be avoided.

In the XLISP-PLUS implementation, each package consists of two object arrays, one for the "internal" symbols and the other for the "external" symbols. External symbols are those intended for access in other packages, while internal symbols are those intended to be private in the package. There is also a "use" list, and a "shadowing symbols" list. In addition, each symbol has a pointer to its "home package." Packages are uniquely named, and have a name space separate from that of symbols. Each package has a "nickname" list of alternate names for itself.

Initially there are three packages, XLISP, KEYWORD, and USER. All keywords are in the external table of the KEYWORD package, while all other symbols are in the XLISP package. The USER package is empty. The "current package" is bound to the symbol *PACKAGE*. The prompt line shows the current package, if it is other than USER, which is the initial default.

When a symbol name is read (with the READ function or from the prompt line) the symbol is looked up in both the internal and external object arrays of the current package and then in the external object arrays of all packages in the current package's use list. By default, the use list of the package USER contains XLISP.

If the symbol name is not found, it is entered in the internal object array of the current package and the symbol's home package is set to the current package. This is also the operation of the INTERN function, which allows one to intern a symbol in either the current or any specified package.

Getting Information

There are several functions which allow obtaining information about packages. FIND-PACKAGE takes a name string and returns the package of that name. Most functions that take package arguments also take a name string or symbol. The contents of a package object can be examined with PACKAGE-NAME (name string of package), PACKAGE-NICKNAMES (nicknames of package), PACKAGE-OBARRAY (either the external or internal object array, this supercedes *OBARRAY*), PACKAGE-SHADOWING-SYMBOLS (list of shadowing symbols), and PACKAGE-USE-LIST (use list). In addition, PACKAGE-USED-BY-LIST returns the list of packages that have the indicated package in their use lists. LIST-ALL-PACKAGES will return a list of all the packages in the system.

SYMBOL-PACKAGE will return the home package of a symbol. FIND-PACKAGE will look up a symbol (given a name string) in a single package and return the symbol (if found) and an indicator of the symbol being internal, external, or inherited via USE-PACKAGE.

Explicitly Accessing Symbols in Other Packages

It is possible to access symbols in other packages that aren't in the use list of the current package via a fully specified symbol name. The syntax is "packagename:symbolname" for symbols external in the other package or "packagename::symbolname" for symbols internal (or external) in the other package. Note that the colon character can be escaped and then becomes a constituent character; there is a difference between |FOO:BAR| and FOO:BAR. Also be wary of the readtable-case :INVERT option — if the fully qualified name is mixed case, then no inversion is performed on either the package or symbol name.

The use of "::" for internal symbols instead of ":" has the intent to make them more difficult to access. Note that there is no mechanism to prevent access — there are no truly private symbols in a package.

An empty package name is taken to be the KEYWORD package. New keywords are always created as external.

Creating a New Package

The function MAKE-PACKAGE is used to create a new package. However, it is an error to create a package with a name that already exists. Application source files should create a new package only if the package they are using doesn't exist. This can be done using FIND-PACKAGE via:

```
(unless (find-package "FOO")
    (make-package "FOO"))
```

or by using the DEFPACKAGE macro. MAKE-PACKAGE has a keyword argument, :USE, which allows specifying the package use list for the new package. Most packages will at least be using the XLISP intrinsic functions, so will typically be defined:

```
(make-package "FOO" :use '("XLISP"))
```

Notice that the package names are expressed as strings. It's also possible to use a symbol, in which case its print name is used as the package name. You can also use the package object itself, if you have it as the return value of a function.

To add the definitions for the new package, one needs to change the current package to be the new package. This is done with the IN-PACKAGE function:

```
(in-package "FOO")
```

which will change the value of *PACKAGE*. Be aware that definitions that use symbol names which are already visible (such as in the XLISP package) will alter existing definitions. You must be careful not to define external symbols in the XLISP package or any other package in the use list. The package system doesn't protect against this!

Exporting Symbols

Just as the XLISP package has external symbols for it's documented functions, applications will typically want their documented interface symbols to be external. This is done with the EXPORT function. The USE-PACKAGE function is then executed in any package which wishes to use the application package.

The EXPORT function makes certain that each exported symbol name not be visible in any other package which uses this package unless that name is in the shadow list or refers to a different symbol. Since in most cases the package won't be used until it is read (and EXPORTed), this is not a frequent occurrence. However the error created is correctable, and action can be taken at load time.

```
> 'conflict                    ;; Create symbol "conflict"
conflict
> (make-package "FOO" :use '("XLISP")) ;; Create a new package, FOO
#<Package FOO>
> (use-package "FOO")              ;; package USER uses FOO
t
> (in-package "FOO")               ;; add definitions to FOO
#<Package FOO>
FOO> 'conflict                   ;; create symbol "conflict" in FOO
conflict
FOO> (export 'conflict)          ;; Now try to export it
Name conflict with user::conflict in #<Package USER>
  when exporting conflict from #<Package FOO>
error: name conflict
if continued: recheck for conflicts
FOO 1>
```

One now has to resolve the conflict. This can be done by not exporting the symbol (which probably isn't desired), uninterning the symbol user::conflict making foo:conflict accessible in USER, or shadowing the symbol user::conflict making foo:conflict only accessible in USER if a fully qualified name is used. If we unintern:

```
FOO 1> (unintern 'user::conflict "USER") ;; unintern the symbol from USER
t
FOO 1> (continue)
[ continue from break loop ]
t
FOO> (symbol-package 'user::conflict)    ;; see what user::conflict is
#<Package FOO>
```

note that it is important in this case to unintern the correct symbol from the correct package. The name conflict message hints at what is needed. Uninterning a symbol removes it from either the internal or external object array in the indicated package. It returns t if successful. If the symbol is not in the object array of the package, then UNINTERN returns nil.

The use of shadowing will be described in a later section.

Typically all symbols that are desired to be exported are done with a single statement

at the top of the application source file:

```
(export '(foo bar bas ...))
```

Once a symbol has been exported, it can be accessed with just a single colon in its fully qualified name. A package can make all external symbols visible without use of colon syntax with the USE-PACKAGE function. The use-package function does not alter any of the object arrays, but adds the used package to the use list of the using package. Again, a name consistency check is performed. In this example, we will do all our operations in the USER package:

```
> 'conflict                ;; add symbol “conflict” to USER package
conflict
> (make-package "FOO")      ;; create a new package, FOO
#<Package FOO>
> 'foo::conflict            ;; add symbol “conflict” to FOO package
foo::conflict
> (export 'foo::conflict "FOO") ;; export “conflict” from FOO
t
> (use-package "FOO")       ;; try to use package FOO in USER
Name conflict of foo:conflict and conflict
  when using #<Package FOO> in #<Package USER>
error: name conflicts
if continued: recheck for conflicts
1>
```

Just like the previous example of EXPORT, we need to pick which symbol named “conflict” will be visible. Again, we can use unintern to remove user::conflict, allowing access of foo:conflict:

```
1> (unintern 'conflict)
t
1> (continue)
[ continue from break loop ]
t
> (symbol-package 'conflict)
#<Package FOO>
```

We didn’t need to specify package USER in the UNINTERN function because the current package is USER and UNINTERN, like most package functions, defaults to the current package.

In this example, the conflict was between a symbol in the package we want toa use with one in the current package. It is also possible to have a conflict between the symbol in the package we want to use and one in another package that has been made visible by a previous USE-PACKAGE. In this case, UNINTERN won’t work because the symbol is not interned in the current package. The solution is to use the function SHADOWING-IMPORT, which is defined in the section “Shadowing Symbols.”

Importing Symbols

Besides the EXPORT/USE-PACKAGE method of making symbols visible in a package, it is also possible to actually intern a symbol that is already interned in another package. The function that does this is IMPORT.

```
> (make-package "FOO")      ;; Make a new package, FOO
#<Package FOO>
> (setq foo::x 10)          ;; Define a symbol in that package
10
> (import 'foo::x)          ;; Import it into the current package
t
> (symbol-package 'x)       ;; Check — is it accessable?
#<Package FOO>              ;; Yes, it's the one in package FOO
```

As with the EXPORT/USE-PACKAGE method, a conflict will occur if the imported symbol already exists in the package being imported into. The problem can be resolved by either using UNINTERN or SHADOWING-IMPORT.

Shadowing Symbols

When a symbol is "shadowed" in a package, conflict errors are ignored. This means that when the package is current that symbol will always be found in a lookup rather than a symbol in another package.

Back to the example in "Exporting Symbols":

```
FOO 1> (unintern 'user::conflict "USER") ;; unintern the symbol from USER
t
FOO 1> (continue)
[ continue from break loop ]
t
FOO> (symbol-package 'user::conflict)    ;; see what user::conflict is
#<Package FOO>
```

Here is result of using shadow:

```
FOO 1> (shadow 'user::conflict "USER") ;; shadow user::conflict in USER
t
FOO 1> (continue)
[ continue from break loop ]
t
FOO> (symbol-package 'user::conflict)   ;; see what user::conflict is
#<Package USER>
```

Consider this example, in which we will do all our actions in the USER package:

```
> (make-package "FOO")        ;; package FOO won't use any other package
#<Package FOO>
> 'foo::car                   ;; this will intern a symbol CAR in FOO
foo::car
> (export 'foo::car "FOO")    ;; export foo::car from FOO
t
> (use-package "FOO")         ;; create an error on USE-PACKAGE
Name conflict of foo:car and car
  when using #<Package FOO> in #<Package USER>
error: name conflicts
if continued: recheck for conflicts
1> (symbol-package 'car)
#<Package XLISP>
1>
```

The conflict occurs between the symbol CAR in FOO and the symbol CAR in XLISP, which is visible in USER because XLISP is in USER's use list. We will have to use SHADOWING-IMPORT to indicate the one we wish to see.

```
1> (shadowing-import 'xlisp:car)
t
1> (continue)
[ continue from break loop ]
t
> (symbol-package 'car)          ;; The visible symbol is the one in XLISP
#<Package XLISP>
> (find-symbol "CAR" "USER")     ;; However it is interned in USER as well!
car
:internal
```

USING MACROS

Macros in Lisp provide a very powerful and flexible method of extending Lisp syntax. They are much, much more powerful than #define macros in C: Lisp macros are a full-fledged code-generation system, while C #define macros are simple string substitutions. Although extremely powerful and useful, macros are also significantly harder to design and debug than normal Lisp functions, and are normally considered a topic for the advanced Lisp developer.

Lisp functions take Lisp values as input and return Lisp values. They are executed at run-time. Lisp macros take Lisp code as input, and return Lisp code. They are executed at compiler pre-processor time, just like in C. The resultant code gets executed at run-time. Almost all the errors that result from using macros can be traced to a misunderstanding of this fact.

Basic Idea

Macros take unevaluated Lisp code and return a Lisp form. This form should be code that calculates the proper value. Example:

```
(defmacro Square (X)   '(* ,X ,X))
```

This means that wherever the pre-processor sees (Square XXX) to replaces it with (* XXX XXX). The resultant code is what the compiler sees.

Debugging technique: macroexpand-1

When designing a function, you can type a call to the function into the Lisp Listener (prompt), and see if it returns the correct value. However, when you type a macro "call" into the Lisp Listener, two things happen: first, the macro is expanded into its resultant code, and then that code is evaluated. It is more useful during debugging to be able to examine the results of these two steps individually. The function macroexpand-1 returns the result of stage one of this process:

```
(macroexpand-1 '(Square 9)) ==> (* 9 9)
```

"If in doubt, macroexpand-1 it out."

Purpose: To control evaluation of the arguments

Since macros are so much harder to use than functions, a good rule of thumb is: *don't use defmacro if defun will work fine*. So, for example, there would be no reason to try to use a macro for Square: a function would be much easier to write and test. In Lisp, unlike in C, there is no need to use macros to avoid the very small runtime overhead of a function call: there is a separate method for that (the "inline" proclamation) that lets you do this without switching to a different syntax. What macros can do that functions cannot is to control when the arguments get evaluated. Functions evaluate all of their arguments before entering the body of the function. Macros don't evaluate any of their arguments at preprocessor time unless you tell it to, so it can expand into code that might not evaluate all of the arguments. For example, suppose that cond was in the language, but if wasn't, and you wanted to write a version of if using cond.

```
(defun Iff-Wrong (Test Then &optional Else)
  (cond
    (Test Then)
    (t    Else)))
```

The problem with this is that it always evaluates all of its arguments, while the semantics of if dictate that exactly one of the Then and Else arguments gets evaluated. For example:

```
(let ((Test 'A))
  (Iff-Wrong (numberp Test)
          (sqrt Test)
          "Sorry, SQRT only defined for numbers"))
```

will crash, since it tries to take (sqrt 'A). A correct version, with behavior identical to the built-in if (except that the real if only has one required arg, not two), would be:

```
(defmacro Iff (Test Then &optional Else)
  "A replacement for IF, takes 2 or 3 arguments. If the first evaluates to   non-NIL, evaluate and return the second.
Otherwise evaluate and return   the third (which defaults to NIL)"
  '(cond
    (,Test ,Then)
    (t     ,Else)) )
```

A similar example would be NAND ("Not AND"), which returns true if at least one of the arguments is false, but, like the built-in and, does "short-circuit evaluation" whereby once it has the answer it returns immediately without evaluating later arguments.

```
(defmacro Nand (&rest Args)
  '(not (and ,@Args)))
```

Bugs

- (A) Trying to evaluate arguments at run-time
- (B) Evaluating arguments too many times
- (C) Variable name clashes.

(A) Trying to evaluate arguments at run-time

Macros are expanded at compiler pre-processor time. Thus, the values of the arguments are generally not available, and code that tries to make use of them will not work. I.e. consider the following definition of Square, which tries to replace (Square 4) with 16 instead of with (* 4 4).

```
(defmacro Square (X)   (* X X))
```

This would indeed work for (Square 4), but would crash for (Square X), since X is probably a variable whose value is not known until run-time. Since macros do sometimes make use of variables and functions at expansion time, and to simplify debugging in general, *it is strongly recommended that all macro definitions and any variables and functions that they use at expansion time (as opposed to code they actually expand into) be placed in a separate file that is loaded before any files containing code that makes use of the macros.*

(B) Evaluating arguments too many times

Let's take another look at our first definition of the Square macro.

```
(defmacro Square (X) '(* ,X ,X))
```

This looks OK on first blush. However, try macroexpand-1'ing a form, and you notice that it evaluates its arguments twice:

```
(macroexpand-1 '(Square (Foo 2))) ==> (* (Foo 2) (Foo 2))
```

Foo gets called twice, but it should only be called once. Not only is this inefficient, but could return the wrong value if Foo does not always return the same value. I.e. consider Next-Integer, which returns 1 the first time called, then 2, then 3. (Square (Next-Integer)) would return N*(N+1), not N², plus would advance N by 2. Similarly, (Square (random 10)) would not necessarily generate a perfect square! With Lisp you have the full power of the language available at preprocessor time (unlike in C), so you can use ordinary Lisp constructs to solve this problem. In this case, let can be used to store the result in a local variable to prevent multiple evaluation. There is no general solution to this type of problem in C.

```
(defmacro Square2 (X)
  '(let ((Temp ,X))
     (* Temp Temp)))

(macroexpand-1 '(Square2 (Foo 2)))
 ==> (let ((Temp (Foo 2)))
       (* Temp Temp))
```

This is what we want.

(C) Variable name clashes.

When using let to suppress multiple evaluation, one needs to be sure that there is no conflict between the local variable chosen and any existing variable names. The above version of Square2 is perfectly safe, but consider instead the following macro, which takes two numbers and squares the sum of them:

```
(defmacro Square-Sum (X Y)
  '(let* ((First ,X)
          (Second ,Y)
          (Sum (+ First Second)))
     (* Sum Sum)) )
```

This looks pretty good, even after macroexpansion:

```
(macroexpand-1 '(Square-Sum 3 4))
 ==> (LET* ((FIRST 3)
            (SECOND 4)
            (SUM (+ FIRST SECOND)))
     (* SUM SUM))
```

which gives the proper result. However, this version has a subtle problem. The local variables we chose would conflict with existing local variable names if a variable named First already existed. E.g.

```
(macroexpand-1 '(Square-Sum 1 First))
 ==> (LET* ((FIRST 1)
            (SECOND FIRST)
            (SUM (+ FIRST SECOND)))
     (* SUM SUM))
```

The problem here is that (SECOND FIRST) gets the value of the new local variable FIRST, not the one you passed in. Thus

```
(let ((First 9)) (Square-Sum 1 First))
```

returns 4, not 100! Solutions to this type of problem are quite complicated, and involve using gensym to generate a local variable name that is guaranteed to be unique.

Moral: even seemingly simple macros are hard to get right, so don't use macros unless they really add something. Both Square and Square-Sum are inappropriate uses of macros.

```
(defmacro Square-Sum2 (X Y)
  (let ((First (gensym "FIRST-"))
        (Second (gensym "SECOND-"))
        (Sum (gensym "SUM-")))
    '(let* ((,First ,X)
            (,Second ,Y)
            (,Sum (+ ,First ,Second)))
       (* ,Sum ,Sum)) ))
```

Now

```
(macroexpand-1 '(Square-Sum2 1 First))
 ==> (LET* ((#:FIRST-590 1)
            (#:SECOND-591 FIRST)
            (#:SUM-592 (+ #:FIRST-590 #:SECOND-591)))
      (* #:SUM-592 #:SUM-592))
```

This expansion has no dependence on any local variable names in the macro definition itself, and since the generated ones are guaranteed to be unique, is safe from name collisions.

USING FILE I/O FUNCTIONS

Input from a File

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

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

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

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

```
(read fp)
```

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

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

```
(close fp)
```

Evaluating this expression will cause the file to be closed.

Output to a File

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

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

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

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

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

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

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

```
(close fp)
```

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

A Slightly More Complicated File Example

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

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

The file will be closed with the next garbage collection.

XLISP-PLUS INTERNALS

Who should read this section?

Anyone poking through the C implementation of XLISP for the first time. This is intended to provide a rough road map of the global XLISP structures and algorithms. If you just want to write Lisp code in XLISP, you don't need to read this section. If you want to tinker with the XLISP implementation code, you should *still* read that before reading this. The following isn't intended to be exhaustively precise -- that's what the source code is for! It is intended only to give you sufficient orientation to give you a fighting chance to understand the code the first time through, instead of the third time.

At the end of the section you'll find an example of how to add new primitive functions to XLISP.

General note on MS-DOS Memory Models

If you are not using MS-DOS ignore all the "FAR" and "NEAR" keywords. Also ignore MEDMEM, and most uppercase macros default to the lowercase library functions.

For MS-DOS, originally XLISP used Large memory model. However in order to allow for a clean MS Windows port (never completed :-() and to improve performance, the code was overhauled to use Medium memory model. Every XLISP object is allocated from FAR memory, so pointers need to be cast to FAR. Local variables, and statically allocated global variables are NEAR, however. Since FAR strings cannot be handled by Medium model library string routines, they must be modeled into a NEAR buffer first.

Other added confusions

There are two separate sets of memory allocation/garbage collection routines. One pair, xldmem/xlimage, was the traditional XLISP allocator. The second pair, dldmem/dlimage, additionally manages arrays and strings. In the original version, malloc and free are used, which can cause memory fragmentation problems. The dl pair compresses array memory segments to eliminate fragmentation.

There are many compilation options. Thank goodness many more have been eliminated.

What is an LVAL?

An LVAL is the C type for a generic pointer to an XLISP garbage-collectable something. (Cons cell, object, string, closure, symbol, vector, whatever.) Virtually every variable in the interpreter is an LVAL. Cons cells contain two LVAL slots, symbols contains four LVAL slots, etc.

What is the obarray?

The obarray is the XLISP symbol table. More precisely, it is a hash table mapping ASCII strings (symbol names) to symbols. ("obarray" is a misnomer, since it contains only XLISP SYMBOLs, and in particular contains no XLISP OBJECTs.) It is used when converting Lisp expressions from text to internal form. Since it is a root for the garbage collector, it also serves to distinguish permanent global-variable symbols from other symbols -- you can permanently protect a symbol from the garbage collector by entering it into the obarray. This is called "interning" the symbol. The obarray is called "obarray" in C and "*OBARRAY*" in XLISP.

When the package facility is compiled, *OBARRAY* no longer exists, and obarray is a list of packages. A package is a structure (an array) of six elements. The first is an obarray for the internal symbols in the package, the second is an obarray for the external symbols, the third is a list of shadowing symbols, the fourth is a list of packages this package uses, the fifth is a list of packages which use this package, and the sixth is a list of the package's names with all but the first being the nicknames. In addition, symbols have an additional attribute in their structure which is the home package of the symbol.

The Interpreter Stacks

XLISP uses two stacks, an "evaluation stack" and an "argument stack". Both are roots for the garbage collector. The evaluation stack is largely private to the interpreter and protects internal values from garbage collection, while the argument stack holds the conventional user-visible stack frames.

The evaluation stack is an EDEPTH-long array of LVAL statically allocated. It grows zeroward.

xlstkbase points to the zero-near end of the evaluation stack.

xlstktop points to the zero-far end of the evaluation stack: the occupied part of the stack lies between xlstack and xlstktop. Note that xlstktop is *NOT* the top of the stack in the conventional sense of indicating the most recent entry on the stack: xlstktop is a static bounds pointer which never changes.

xlstack starts at the zero-far end of the evaluation stack. *xlstack is the most recent LVAL on the stack. The garbage collector MARKs everything reachable from the evaluation stack (among other things), so we frequently push things on this stack while C code is manipulating them. (Via xlsave(), xlprotect(), xlsave1(), xlprot1().)

The argument stack is an ADEPTH-long array of LVAL. It also grows zeroward. The evaluator pushes arguments on the argument stack at the start of a function call (form evaluation). Built-in functions usually eat them directly off the stack. For user-Lisp functions xleval.c:evfun() pops them off the stack and binds them to the appropriate symbols before beginning execution of the function body proper.

xlargstkbase is the zero-near end of argument stack. xlargstktop is the zero-far end of argument stack. Like xlstktop, xlargstktop is a static bounds pointer.

*xlsp ("sp"=="stack pointer") is the most recent item on the argument stack.

xlfp ("fp"=="frame pointer") is the base of the current stackframe.

What is a context?

An XLISP "context" is something like a checkpoint, recording a particular point buried in the execution history so that we can abort/return back to it. Contexts are used to implement call/return, catch/throw, signals, goto's, and breaks. xlcontext points to the chain of active contexts, the top one being the second-newest active context. (The newest -- that is, current -- active context is implemented by the variables xlstack xlenv xlfenv xldenv xlcontext xlargv xlargc xlfp xlsp.) Context records are written by xljump.c:xlbegin() and read by xljump.c:xljump(). Context records are C structures on the C program stack; They are not in the dynamic memory pool or on the Lisp execution or argument stacks.

To create a context, the function defines a local CONTEXT structure, and then calls xlbegin, passing the address of the structure, the appropriate context flags (ORed together, if more than one), and an expression which is the tag for return and throw contexts, and NIL for others. xlend is used to end the context. Within the context, a setjmp using the jump buffer in the context structure provides the mechanism to return to the function.

The xljump function takes three arguments, the target context, a mask value (which is returned by the setjmp in the target context), and a return value. The return value is typically NIL, but holds the return value of RETURN and THROW functions for those operations. xljump unwinds the contexts until the target context is found, or an UNWIND-PROTECT context is found. In the latter case, xunwindprotect saves the unwind target so that unwinding can resume.

The implemented context flags are:

- CF_GO -- used by TAGBODY, searched by GO (and the xlgo function).
- CF_RETURN -- used by named blocks and functions, searched by RETURN.
- CF_THROW -- used by CATCH, searched by THROW.

- CF_ERROR -- used by ERRSET to mark context of an error handler.
- CF_UNWIND -- used by UNWIND-PROTECT.
- CF_TOPLEVEL -- used at the top-level loop.
- CF_BRKLEVEL -- used in top level and break loops. Searched on uncaught (with CF_ERROR) errors.
- CF_CLEANUP -- used in the top level and break loops. Searched when going back a break level. Note that this and CF_BRKLEVEL always appear together in contexts, but imply different functionality when signaled.
- CF_CONTINUE -- used in break loops to cause continuation.

In most cases, the mask value passed to xljump is the context flag. For the break loop, for example, this allows dispatching based on the flag value of BRKLEVEL (which re-enters the debug loop), CLEANUP (which returns to the previous break loop), or CONTINUE (which attempts to continue execution). Starting with XLISP2.1g, in the case of the TAGBODY context, the mask value is the index into the TAGBODY of the jump target.

What is an environment?

An environment is basically a store of symbol-value pairs, used to resolve variable references by the Lisp program. XLISP maintains three environments, in the global variables xlenv, xlfenv and xldenv.

xlenv and xlfenv are linked-list stacks which are pushed when we enter a function and popped when we exit it. We also switch xlenv + xlfenf environments entirely when we begin executing a new closure (user-fn written in Lisp). The xlenv environment is enlarged during a LET function (or other special form with value bindings), while the xlfenv environment is enlarged with FLET/MACROLET/LABELS.

The xlenv environment is the most heavily used environment. It is used to resolve everyday data references to local variables. It consists of a list of frames (and objects). Each frame is a list of sym-val pairs. In the case of an object, we check all the instance and class variables of the object, then do the same for its superclass, until we run out of superclasses. If the symbol is not found it has not been bound and its global value is used.

The xlfenv environment is used to find function values instead of variable values.

When we send a message, we set xlenv to the value it had when the message CLOSURE was built, then push on (obj msg-class), where msg-class is the [super]class defining the method. (We also set xlfenv to the value xlfenv had when the method was built.) This makes the object instance variables part of the environment, and saves the information needed to correctly resolve references to class variables, and to implement SEND-SUPER.

The xldenv environment tracks the old values of global variables which we have changed but intend to restore later to their original values, particularly for variables (symbols) marked as F_SPECIAL, but also for progv. This is also used internally when we bind and unbind s_evalhook and s_applyhook (*EVALHOOK* and *APPLYHOOK*). It is a simple list of sym-val pairs, treated as a stack.

These environments are manipulated in C via the XLISP.h macros xlframe(e), xlbind(s,v), xlfbind(s,v), xlpbind(s,v,e), xldbind(s,v), xlunbind(e).

How are multiple return values handled?

The first value is always returned via the C function mechanism. All return values are passed back in an array xlresults[]. The number of return values are specified in xlnumresults. In order to avoid having to repeat the multiple value code in every function, a flag in the SUBR or FSUBR cell indicates if multiple values are used, and code in xleval.c mimics multiple value functions for single value functions. The function VALUES, defined in xvalues() allows closures to return multiple values.

How are XLISP entities stored and identified?

Conceptually, XLISP manages memory as a single array of fixed-size objects. Keeping all objects the same size simplifies memory management enormously, since any object can be allocated anywhere, and complex compacting schemes aren't needed. Every LVAL pointer points somewhere in this array. Every XLISP object has the basic format (xldmem.h:typdef struct node)

struct node { char n_type; char n_flags; LVAL car; LVAL cdr; }

where n_type is one of:

- FREE A node on the free list.
- SUBR A function implemented in C. (Needs evaluated arguments.)
- FSUBR A special function implemented in C. (Needs unevaluated arguments).
- CONS A regular Lisp cons cell.
- FIXNUM An integer.
- FLONUM A floating-point number.
- STRING A string.
- STREAM An input or output file.
- CHAR An ASCII character.
- USTREAM An internal stream.
- RATIO A ratio.
- SYMBOL A symbol.
- OBJECT Any object, including class objects.
- VECTOR A variable-size array of LVALs.
- CLOSURE Result of DEFUN or LAMBDA -- a function written in Lisp.
- STRUCT A structure.
- COMPLEX A complex number
- BIGNUM A bignum (integer that won't fit a FIXNUM cell)
- PACKAGE A package

Messages may be sent only to nodes with n_type == OBJECT.

Obviously, several of the above types won't fit in a fixed-size two-slot node. The escape is to have them malloc() some memory and have one of the slots point to it -- VECTOR is the archetype. For example, see xldmem.c:newvector(). To some extent, this malloc() hack simply exports the memory- fragmentation problem to the C malloc()/free() routines. However, it helps keep XLISP simple, and it has the happy side-effect of unpinning the body of the vector, so that vectors can easily be expanded and contracted. When the dldmem.c version of the memory manager is used, this memory is managed by XLISP and vector memory is allocated from compressible vector segments.

The garbage collector has special-case code for each of the above node types, so it can find all LVAL slots and recycle any malloc()ed ram when a node is garbage-collected. If the collected node is a STREAM, then it's associated file is closed.

XLISP pre-allocates nodes for all ASCII characters, and for small integers. These nodes are never garbage-collected.

As a practical matter, allocating all nodes in a single array is not very sensible. Instead, nodes are allocated as needed, in segments of one or two thousand nodes, and the segments linked by a pointer chain rooted at xldmem.c:segs.

How are vectors implemented?

An XLISP vector is a generic array of LVAL slots. Vectors are also the canonical illustration of XLISP's escape mechanism for node types which need more than two LVAL slots (the maximum possible in the fixed-size nodes in the dynamic memory pool). The node CAR/CDR slots for a vector hold a size field plus a pointer to a malloc()ed ram chunk, which is automatically free()ed when the vector is garbage-collected.

xldmem.h defines macros for reading and writing vector fields and slots: getsize(), getelement() and setelement(). It also defines macros for accessing each of the other types of XLISP nodes.

How are strings implemented?

Strings work much like vectors: The node has a pointer to a malloc()ed ram chunk which is automatically free()ed when the string gets garbage-collected. The size field of a string is size in bytes. Unlike C, the null character can be a string constituent.

How are various numeric types implemented?

There are five numeric types: FIXNUM, BIGNUM, RATIO, FLONUM, and COMPLEX. The first two constitute integers, the first three rational numbers. BIGNUM, RATIO, and COMPLEX are optional types, with BIGNUM and RATIO requiring COMPLEX.

Numeric cells are never modified, new cells are created as necessary.

FIXNUMs are integers that fit in a FIXTYPE (typically "long"). The car slot (basically) contains the number. There is a pool of preallocated small fixnums (range SFIXMIN to SFIXMAX) that get reused. Only these small FIXNUMs can be tested for equality with EQ.

BIGNUMs are used when an integer will not fit in a FIXNUM. A BIGNUM structure is like an character string, however the length is a multiple of sizeof(unsigned short). The first unsigned short cell contains the sign of the number, and the remaining unsigned short cells contain the number, most significant part first. The first numeric cell may be zero. Functions that can create bignum results will convert automatically to FIXNUM if the result fits, however with the function small BIGNUMs can exist, and the number of numeric cells must be at least two.

RATIOs have the car field pointing to the numerator cell and the cdr field pointing to the denominator cell. These values must be either both FIXNUM or both BIGNUM.

FLONUMs have the car (overflowing into the cdr) field containing the floating point value.

COMPLEXs were implemented as a two element array, but now are implemented like ratios. The two elements are either both FLONUMs or both integers (rationals if ratios/bignums allowed).

How are streams implemented?

A file stream node has four fields: a file pointer (type FILEP), a saved lookahead character, flags, and character position within a line.

The file pointer is either a FILE * or the index into the file table. In the preferred file table implementation, the index values CLOSED, STDIN, STDOUT, and CONSOLE are -1, and the first three entries which are forced to stdin, stdout, and stderr, respectively. The file table entry contains the FILE *, the actual file name (full pathname), and for Windows the reopen mode and file position. Under Windows, all files are closed when waiting for input in order to be "friendly" to the Windows environment. The extra information allows closing the files and then reopening them exactly as they were.

The saved lookahead character allows the peekchar function and internal unreadchar function to work. The value is set to 0 when there is no lookahead character. This does not pose a problem since ASCII files do not have null characters.

The flags field tells if the file is open for reading, writing, or both, and also what the last direction was so that an fseek can be performed on direction change. An additional bit keeps track of the file being opened for ASCII or BINARY access.

The character position is used for tab calculations.

An unnamed stream is simply a TCONC structure. None of the above file information applies to unnamed streams.

How are symbols implemented?

A symbol is a generic user-visible Lisp variable, with separate slots for print name, value, function, and property list. Any or all of these slots (including name) may be NIL. You create a symbol in C by calling "xlmakesym(name)" or "xlenter(name)" (to make a symbol and enter it in the obarray). You create a symbol in XLISP with the READ function, or by calling "(gensym)", or indirectly in various ways. Most of the symbol-specific code in the interpreter is in xlsym.c.

Physically, a symbol is implemented like a four- or five-slot vector. A couple of free bits in the node structure are used as flags for F_SPECIAL (special variables) and F_CONSTANT (constants).

A symbol is marked as unbound (no value) or undefined (no function) by placing a pointer to symbol s_unbound in the value or function field, respectively. The symbol s_unbound is not interned and is not used other than to set and check for unbound variables and undefined functions.

The symbol NIL is statically allocated so its address is a constant. This makes the frequent comparison of a pointer to NIL faster and more compact (in some cases).

Random musing: Abstractly, the Lisp symbols plus cons cells (etc.) constitute a single directed graph, and the symbols mark spots where normal recursive evaluation should stop. Normal Lisp programming practice is to have a symbol in every cycle in the graph, so that recursive traversal can be done without MARK bits.

How are closures implemented?

A closure, the return value from a lambda, is a regular coded-in-Lisp fn. Physically, it implemented like an eleven-slot vector, with the node n_type field hacked to contain CLOSURE instead of VECTOR. The vector slots contain:

name symbol -- 1st arg of DEFUN. NIL for LAMBDA closures. type (s_lambda or s_macro). args List of "required" formal arguments (as symbols) oargs List of "optional" args, each like: (name (default specified-p)) rest Name of "&rest" formal arg, else NIL. kargs keyword args, each like: ((':foo 'bar default specified-p)) aargs &aux vars, each like: (('arg default)) body actual code (as Lisp list) for fn. env value of xlenv when the closure was built. NIL for macros. fenv value of xlfenv when the closure was built. NIL for macros. lambda The original formal args list in the defining form.

The lambda field is for printout purposes. The remaining fields store a pre-digested version of the formal args list. This is a limited form of compilation: by processing the args list at closure-creation time, we reduce the work needed during calls to the closure.

How are objects implemented?

An object is implemented like a vector, with the size determined by the number of instance variables. The first slot in the vector points to the class of the object; the remaining slots hold the instance variables for

the object. An object needs enough slots to hold all the instance variables defined by its class, *plus* all the instance variables defined by all of its superclasses.

How are classes implemented?

A class is a specific kind of object, hence has a class pointer plus instance variables. All classes have the following instance variables:

- MESSAGES A list of (interned-symbol method-closure) pairs.
- IVARS Instance variable names: A list of interned symbols.
- CVARS Class variable names: A list of interned symbols.
- CVALS Class variable values: A vector of values.
- SUPERCLASS A pointer to the superclass.
- IVARCNT Number of class instance variables, as a fixnum.
- IVARTOTAL Total number of instance variables, as a fixnum.
- PNAME Printname for this class.

IVARCNT is the count of the number of instance variables defined by our class. IVARTOTAL is the total number of instance variables in an object of this class -- IVARCNT for this class plus the IVARCNTs from all of our superclasses.

How is the class hierarchy laid out?

The fundamental objects are the OBJECT and CLASS class objects. (Both are instances of class CLASS, and since CLASSes are a particular kind of OBJECT, both are also objects, with n_type==OBJECT. Bear with me!)

OBJECT is the root of the class hierarchy: everything you can send a message to is of type OBJECT. (Vectors, chars, integers and so forth stand outside the object hierarchy -- you can't send messages to them. I'm not sure why Dave did it this way.) OBJECT defines the messages:

:isnew -- Does nothing.

:class -- Returns contents of class-pointer slot.

:show -- Prints names of obj, obj->class and instance vars (for debugging).

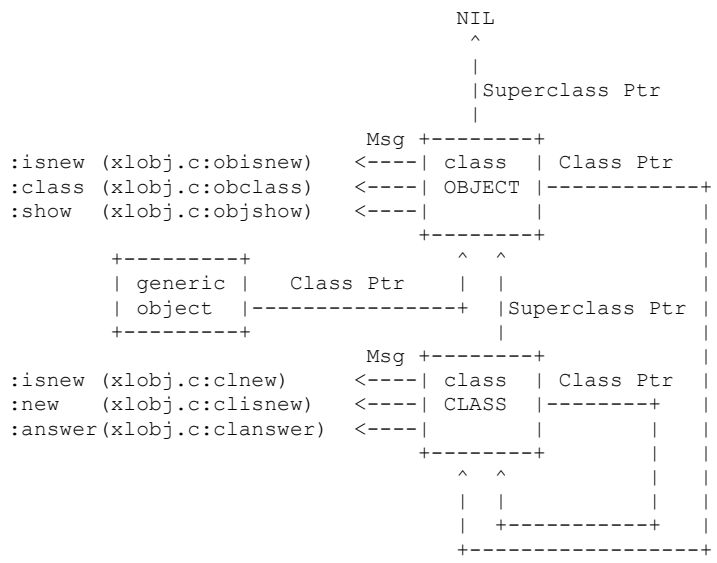
:prin1 -- print the object to argument stream

A CLASS is a specialized type of OBJECT (with instance variables like MESSAGES which generic OBJECTs lack), class CLASS hence has class OBJECT as its superclass. The CLASS object defines the messages:

- :new -- Create new object with self.IVARTOTAL LVAR slots, plus one for the class pointer. Point class slot to self. Set new.n_type char to OBJECT.
- :isnew -- Fill in IVARS, CVARS, CVALS, SUPERCLASS, IVARCNT and IVARTOTAL, using parameters from :new call. (The :isnew msg inherits the :new msg parameters because the :isnew msg is generated automatically after each :new msg, courtesy of a special hack in xlobj.c:sendmsg().)
- :answer -- Add a (msg closure) pair to self.MESSAGES.

Here's a figure to summarize the above, with a generic object thrown in for good measure. Note that all instances of CLASS will have a SUPERCLASS pointer, but no non-class object will. Note also that the messages known to an object are those which can be reached by following exactly one Class Ptr and then zero or more Superclass Ptrs. For example, the generic object can respond to :ISNEW, :CLASS and

:SHOW, but not to :NEW or :ANSWER. (The functions implementing the given messages are shown in parentheses.)



Thus, class CLASS inherits the :CLASS and :SHOW messages from class OBJECT, overrides the default :ISNEW message, and provides new the messages :NEW and :ANSWER.

New classes are created by (send CLASS :NEW …) messages. Their Class Ptr will point to CLASS. By default, they will have OBJECT as their superclass, but this can be overridden by the second optional argument to :NEW.

The above basic structure is set up by xlobj.c:xloinit().

How do we look up the value of a variable?

When we're cruising along evaluating an expression and encounter a symbol, the symbol might refer to a global variable, an instance variable, or a class variable in any of our superclasses. Figuring out which means digging through the environment. The canonical place this happens is in xleval.c:xleval(), which simply passes the buck to xlsym.c:xlgetvalue(), which in turn passes the buck to xlxsym.c:xlxgetvalue(), where the fun of scanning down xlenv begins. The xlenv environment looks something like

Backbone Environment frame contents

- xlenv --> frame ((sym val) (sym val) (sym val) …)
- frame …
- object (obj msg-class)
- frame …
- object …
- frame …
- …

The "frame" lines are due to everyday nested constructs like LET expressions, while the "object" lines represent an object environment entered via a message send. xlxgetvalue scans the environment left to right, and then top to bottom. It scans down the regular environment frames itself, and calls xlobj.c:xlobjgetvalue() to search the object environment frames.

xlobjgetvalue() first searches for the symbol in the msg-class, then in all the successive superclasses of msg-class. In each class, it first checks the list of instance-variable names in the IVARS slot, then the list of class-variables name in the CVARS slot.

How are function calls implemented?

xleval.c contains the central expression-evaluation code. xleval.c:xleval() is the standard top-level entry point. The two central functions are xleval.c:xlevform() and xleval.c:evfun(). xlevform() can evaluate four kinds of expression nodes:

SUBR: A normal primitive fn coded in C. We call evpushargs() to evaluate and push the arguments, then call the primitive.

FSUBR: A special primitive fn coded in C, which (like IF) wants its arguments unevaluated. We call pushargs() (instead of evpushargs()) and then the C fn.

CLOSURE: A pre-processed written-in-Lisp fn from a DEFUN or LAMBDA. We call evpushargs() and then evfun().

CONS: We issue an error if it isn't a LAMBDA, otherwise we call xleval.c:xlclose() to build a CLOSURE from the LAMBDA, and fall into the CLOSURE code.

The common thread in all the above cases is that we call evpushargs() or pushargs() to push all the arguments on the evaluation stack, leaving the number and location of the arguments in the global variables xlargc and xlargv. The primitive C functions consume their arguments directly from the argument stack.

xleval.c:evfun() evaluates a CLOSURE by:

(1) Switching xlenv and xlfenv to the values they had when the CLOSURE was built. (These values are recorded in the CLOSURE.)

(2) Binding the arguments to the environment. This involves scanning through the section of the argument stack indicated by xlargc/xlargv, using information from the CLOSURE to resolve keyword arguments correctly and assign appropriate default values to optional arguments, among other things.

(3) Evaluating the body of the function via xleval.c:xleval().

(4) Cleaning up and restoring the original environment.

How are message-sends implemented?

We scan the MESSAGES list in the CLASS object of the recipient, looking for a (message-symbol method) pair that matches our message symbol. If necessary, we scan the MESSAGES lists of the recipients superclasses too. (xlobj.c:sendmsg().) Once we find it, we basically do a normal function evaluation. (xlobjl.c:evmethod().) Two oddities: We need to replace the message-symbol by the recipient on the argument stack to make things look normal, and we need to push an 'object' stack entry on the xlenv environment so we remember which class is handling the message.

How is garbage collection implemented?

The dynamic memory pool managed by XLISP consists of a chain of memory *segments* rooted at global C variable "segs". Each segment contains an array of "struct node"s plus a pointer to the next segment. Each node contains a n_type field and a MARK bit, which is zero except during garbage collection.

XLISP uses a simple, classical mark-and-sweep garbage collector. When it runs out of memory (fnodes==NIL), it does a recursive traversal setting the MARK flag on all nodes reachable from the obarray, the three environments xlenv/xlfenv/xldenv, and the evaluation and argument stacks. (A

"switch" on the n_type field tells us how to find all the LVAL slots in the node (plus associated storage), and a pointer-reversal trick lets us avoid using too much stack space during the traversal.) sweep() then adds all un-MARKed LVALs to fnodes, and clears the MARK bit on the remaining nodes. If this fails to produce enough free nodes, a new segment is malloc()ed.

The code to do this stuff is mostly in xldmem.c.

How is garbage collection of vectors/strings implemented?

In the dldmem.c version, vector/string space is allocated from a memory pool maintained by XLISP, rather than relying on the C library malloc() and free() functions. The pool is a linked list of VSEGMENT with the root vsegments.

When no free memory of a size necessary for an allocation request is available, a garbage collection is performed. If there is still no available memory, then a new vector segment is allocated.

The garbage collection process compacts the memory in each vector segment. This is an easy process because each allocated vector area is pointed to by only one location (in an LVAL), and a back pointer is maintained from the vector segment to the LVAL. Empty vector segments are returned to the heap using free() if there greater than 50% of the vector space is free. This is done to reduce thrashing while making memory available for allocation to LVALs.

How does XLISP initialize?

A major confusing aspect of XLISP is how it initializes, and how save/restore works. This is a multi step process that must take place in a specific order.

When XLISP starts, there is no obarray, and in fact no symbols at all. All initial symbols must be created as part of initialization. In addition the character and small integer cells must be created, and all the C variables that point to XLISP symbols must be initialized.

Initialization is mostly performed in xlinit.c, from the function xlinit(). This function is called from main() after main parses the command line, calls osinit() to initialize the OS interface, and sets the initial top level context. This initial context causes a fatal error to occur if any error happens during the initialization process.

The first step of xlinit() is to call xlminit(), which is in xldmem.c or dldmem.c. This initializes the pointers for the memory manager, stacks, creates the small integer and character LVAL segments, and creates the NIL symbol by filling in the statically allocated NIL LVAL from one that is temporarily created. This first call of xlmakesym will do the first garbage collection -- all of the roots used for the mark routine have been set so that marking will not occur (there is nothing to mark!). It is important, however, that garbage collection not occur again until initialization is completed. This can be assured by having the allocation segment size, NNODES, be large enough for the entire initialization.

The second step in xlinit() is to call xldbug.c:xldinit() to turn debugging off.

At this point, if a restore is to occur from a workspace file, then the restore is attempted. If the restore is successful, then initialization is finished. See "How does save/restore work?" which is the next question.

If the restore fails or there is no file to restore, an initial workspace must be created. This is done by function initwks(), also in xlinit.c.

Initwks() starts by calling four initialization functions.

xlsym.c:xlsinit() creates the symbol *OBARRAY* (and sets the variable obarray to point to it), creates the object array as the value, and enters obarray into the array.

xlsymbols() is called next. It enters all of the symbols referenced by the interpreter into the obarray, and saves their addresses in C variables. This function is also called during restores, so it is important that it

does not change the value of any symbols where the value would be set by restore. If the unbound indicator symbol does not exist, one is created. Then it puts NIL in the obarray if not already there (NIL being already created), then all the other symbols are added (if they don't exist), and their addresses saved in C variables.

This function also initializes constants such as NIL, T, and PI. Because a saved workspace might have a different file stream environment, xlsymbols always initializes the standard file streams based on the current XLISP invocation. It builds the structure property for RANDOM-STATE structures. It then (shudder!) calls two other initialization functions xlobj.c:obsymbols() and ossymbols (in the appropriate *stuff.c file) to enter symbols used in the object feature and operating system specific code, respectively.

Returning to initwks(), two additional initialization routines are called. Xlread.c:xlrinit() initializes the read-table and installs the standard read macros. Xlobj.c:xloinit() creates the class and object objects.

Since the NIL and *OBARRAY* symbols were created before the unbound marker symbol, initwks sets the function values of these symbols, and of the unbound marker symbol, to be unbound. It then initializes all the global variables. Finally it creates all the built-in function bindings from the funtab table. The synonym functions are created last.

How are workspaces saved and restored?

All the work is done in dlimage.c or xlimage.c, depending on the memory management scheme. The basic trick in a save is that memory locations upon a restore would be entirely different. Because of that, addresses written out are converted into offsets from the start of the segs LVAL segment list. In the restore operation, the offset is converted back into an address; if the offset is larger than the allocated segment memory, additional segment memory is allocated until an address can be calculated.

Looking at the save function, xlisave(char *fname), the argument string is taken as the name of a workspace file, and a binary file is created of that name. Then a garbage collection is performed since it would be wasteful to write out garbage.

The size of ftab is written as a validity check, figuring that if the configuration changed, then this value would be different.

The offset of the *obarray* symbol is written next, since obarray is crucial to doing a restore -- it is used to get the addresses of all the other symbols used in the interpreter. Since NIL is a statically allocated symbol, the offsets to its function, property list, and printname are written. (I bet you didn't know you could define a function called NIL!)

Now the segs are traversed and all nodes are written out. The nodes are written in the format of a one byte tag followed by information dependent on the node type. Since many locations in the segment can be empty, one node type, FREE, has data that sets the next offset in the file, thus allowing skipping of many locations with a single command. The function setoffset(), called before writing tags in the other cases, handles writing the FREE entry. CONS, USTREAM, COMPLEX, and RATIO cells consist of two pointers, which are written after conversion into offsets. For all other node types, the raw information is written by writenode(). This could be optimized since not all information is needed (for instance, the address of arrays won't be needed!)

A terminator entry (FREE with length of zero) is written, and the segs are traversed again, looking for nodes with attached array/string data and streams. For the types where the attached data is an array, the array elements (pointers) are converted to offsets and written out. For a string, the characters are written. For a stream (assuming FILETABLE is used), the file's name and position are written if it is other than standard input, output, or error (which cannot be saved or restored).

Restoring a workspace is somewhat more difficult. The file is opened and checked for validity. Then the old memory image is deleted. During the deletion, any open file streams other than standard input, output, or error are closed. All of the global memory allocation pointers and stacks are reset, just like in

initialization. Since the fixnum and character segments are of fixed size and need a known location, there are allocated. Their values, however, will be filled in from the workspace file. This is another wasteful inefficiency, but at least these segments are small.

The global pointer obarray is read from the file. As mentioned earlier, the cviptr() function converts the offset in the file into a physical address, allocating more node segments as necessary. The array portion of the NIL symbol is allocated, and its function, property list, and printname pointers are read from the file.

Then the node information is read in. For type FREE, the offset is adjusted. For CONS, USTREAM, COMPLEX, and (when bignums defined) RATIO the car and cdr pointers are read, for other types, the LVAL data is read raw.

The LVAL segments are scanned, and now the vector/string components of nodes are read. Since the order of nodes is unchanged, the data is read into the correct nodes. For vector types, the size field of the LVAL is consulted, vector space is allocated, and offsets are read from the file and converted into pointers. For strings, the string space is allocated and the string is read. For streams other than standard input, output, or error, the file name and position is read and an attempt is made to open the file. If the file can be opened, then it is positioned.

During the scan, for SUBR/FSUBR nodes funtab is consulted and the correct subroutine address is inserted.

During the whole process, if a tag is invalid or the file size is not correct, a fatal error occurs.

A garbage collection is performed to initialize the free space for future node allocation.

Finally xlsymbols() is called, as in the initialization process, so that the C pointers in the interpreter can be set. This also sets the streams for standard input, output, and error to the correct values.

How do I add a new primitive fn to XLISP?

Add a line to the end of xlftab.c:funtab[], and a function prototype to xlftab.h. This table contains a list of triples:

The first element of each triple is the function name as it will appear to the programmer. Make it all upper case.

The second element is S (for SUBR) if (like most fns) your function wants its arguments pre-evaluated, else F (for FSUBR).

The third element is the name of the C function to call.

Remember that your arguments arrive on the XLISP argument rather than via the usual C parameter mechanism.

CAUTION: Try to keep your files separate from generic XLISP files, and to minimize the number of changes you make in the generic XLISP files. This way, you'll have an easier time re-installing your changes when new versions of XLISP come out. It's a good idea to put a marker (like a comment with your initials) on each line you change or insert in the generic XLISP fileset. For example, if you are going to add many primitive functions to your XLISP, use an #include file rather than putting them all in xlftab.c.

CAUTION: Remember that you usually need to protect the LVAL variables in your function from the garbage-collector. It never hurts to do this, and often produces obscure bugs if you don't. Generic code for a new primitive fn:

```
/* xlsamplefun - do useless stuff. */
/* Called like (samplefun '(a c b) 1 2.0) */
LVAL xlsamplefun()
{
 /* Variables to hold the arguments: */
 LVAL list_arg, integer_arg, float_arg;

 /* Get the arguments, with appropriate errors*/
 /* if any are of the wrong type. Look in     */
 /* XLISP.h for macros to read other types of */
 /* arguments. Look in xlmath.c for examples  */
 /* of functions which can handle an argument */
 /* which may be either int or float: */
 list_arg = xlgalist() ;        /* "XLISP Get A LIST"  */
 integer_arg = xlgafixnum();    /* "XLISP Get A FIXNUM"*/
 float_arg = xlgaflonum();      /* "XLISP Get A FLONUM"*/

 /* Issue an error message if there are any extra arguments: */
 xllastarg();

 /* Call a separate C function to do the actual */
 /* work. This way, the main function can       */
 /* be called from both XLISP code and C code.  */
 /* By convention, the name of the XLISP wrapper*/
 /* starts with "xl", and the native C function */
 /* has the same name minus the "xl" prefix:    */
 return samplefun( list_arg, integer_arg, float_arg );
}
LVAL samplefun( list_arg, integer_arg, float_arg )
LVAL list_arg, integer_arg, float_arg;
{
 FIXTYPE val_of_integer_arg;
 FLOTYPE val_of_float_arg;

 /* Variables which will point to Lisp objects: */
 LVAL result;

 LVAL list_ptr;
 LVAL float_ptr;
 LVAL int_ptr;
 /* Protect our internal pointers by */
 /* pushing them on the evaluation   */
 /* stack so the garbage collector   */
 /* can't recycle them in the middle */
 /* of the routine:                  */
 xlstkcheck(3);    /* Make sure following xlsave  */
                   /* calls won't overrun stack.  */
 xlsave(list_ptr); /* Use xlsave1() if you don't  */
 xlsave(float_ptr);/* do an xlstkcheck().         */
 xlsave(int_ptr);

/* Create an internal list structure, protected   */
/* against garbage collection until we exit fn:   */
```

```
 list_ptr = cons(list_arg,list_arg);

/* Get the actual values of our fixnum and flonum:*/
 val_of_integer_arg = getfixnum( integer_arg );
 val_of_float_arg = getflonum( float_arg );

/*****************************************/
/* You can have any amount of intermediate */
/* computations at this point in the fn... */
/*****************************************/

/* Make new numeric values to return: */
integer_ptr = cvflonum( val_of_integer_arg * 3 );
float_ptr = cvflonum( val_of_float_arg * 3.0 );

/* Cons it all together to produce a return value: */
result = cons( list_ptr, cons( integer_ptr, cons( float_ptr, NIL ) ) );

/* Restore the stack, cancelling the xlsave()s: */
xlpopn(3);
/* Use xlpop() for a single argument.*/

return result; }
```

COMPILATION OPTIONS

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

1. Available Functions (all turned on by default)
   - SRCHFCN supplies SEARCH
   - MAPFCNS supplies SOME EVERY NOTANY NOTEVERY and MAP
   - POSFCNS supplies POSITION-* COUNT-* and FIND-* functions
   - REMDUPS supplies REMOVE-DUPLICATES
   - REDUCE supplies REDUCE
   - SUBSTITUTE supplies SUBSTITUTE-* and NSUBSTITUTE-*
   - ADDEDTAA supplies GENERIC
   - TIMES supplies TIME GET-INTERNAL-RUN-TIME GET-INTERNAL-REAL-TIME and the constant INTERNAL-TIME-UNITS-PER-SECOND
   - RANDOM supplies RANDOM-NUMBER-STATE type, *RANDOM-STATE*, and the function MAKE-RANDOM-STATE. Requires TIMES.
   - HASHFCNS supplies SETHASH MAKE-HASH-TABLE REMHASH MAPHASH CLRHASH and HASH-TABLE-COUNT
   - SETS supplies ADJOIN UNION INTERSECTION SET-DIFFERENCE and SUBSETP
   - SAVERESTORE supplies SAVE and RESTORE
   - GRAPHICS supplies graphic functions (when available)

2. Features (all turned on by default)
   - COMPLX adds complex number support including math functions COMPLEX COMPLEXP IMAGPART REALPART CONJUGATE PHASE LOG FLOOR CEILING ROUND PI LCM and ASH
   - BIGNUMS adds bignum, ratio, and read/print radix support. Requires COMPLX.
   - NOOVFIXNUM Check for fixnum overflow, and convert to flonum (only applies if BIGNUMS not used)
   - PACKAGES uses the packages implementation. Some people find XLISP-PLUS easier to use if this is not defined.
   - MULVALS multiple value returns
   - FILETABLE files referenced via a table -- allows saving and restoring open files (in WKS files), and is required by Microsoft Windows versions. Also allows functions TRUENAME and DELETE-FILE.
   - KEYARG adds :key keyword option to functions having it
   - FROMEND adds the :from-end and :count keywords to functions having them
   - AOKEY makes &allow-other-keys functional. Without this option, all functions behave as though &allow-other-keys is always specified.
   - APPLYHOOK adds applyhook support

3. Backward compatibility
   - OLDERRORS makes CERROR and ERROR work as in XLISP-PLUS 2.1e or earlier, which is not compatible with more recent versions (or Common Lisp).
   - LEXBIND lexical tag scoping for TAGBODY/GO and BLOCK/RETURN, as in Common Lisp. If not defined, then the original dynamic scoping is used.
   - NOCLASSVARINH disables class variable inheritance. The documentation always stated class variables are not inherited, but the implementation always had them inherited.

4. Environmental options
   - ASCII8 eight bit ASCII character support
   - ANSI8 used in addition to ASCII8 for MS Windows character code page
   - READTABLECASE adds *readtable-case* and its functionality
   - PATHNAMES allows environment variable to specify search path for RESTORE and LOAD functions
   - BIGENDIANFILE binary files use "bigendian" orientation. Normally this option is defined when BIGENDIAN (required on bigendian systems) is defined, but this has been made a separate option to allow file portability between systems.

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

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

INDEX