V M L A B S

XLisp Tutorial:

an Introduction for C Programmers

Revision 1.6

6 August 1998

VM Labs, Inc.
167 So. San Antonio Rd, Suite 17
Los Altos, CA 94022
Tel: (415) 917 8050
Fax: (415) 917 8052

CONTENTS

1. INTRODUCTION 5

1.1 XLISP DATA TYPES 5
1.1.1 Numeric data types 5
1.1.2 Non-numeric data types 5

2. THE XLISP EVALUATOR 7

2.1 SPECIAL FORMS 8
2.1.1 DEFINE 8
2.1.2 IF 8
2.2 NESTED ENVIRONMENTS AND LET 9
2.3 SET! 11
2.4 QUOTING 12

3. LAMBDA 14

3.1 CREATION AND MANIPULATION OF LISTS 16
3.1.1 Building up lists with CONS 17
3.1.2 Concatenating lists 18
3.2 PROCEDURE DEFINITION IN XLISP 18
3.2.1 Optional, rest, and keyword arguments 20

4. XLISP FILE I/O 22

5. XLISP OBJECT SYSTEM 25

6. NAMED LET 27

7. MULTIPLE VALUES 29

8. REFERENCES 31

9. INDEX 32

1. INTRODUCTION

XLisp is a lisp implementation written and maintained by David Betz. It is based largely on the standard lisp dialect, Scheme, which it extends with a Smalltalk-like Object System. Generally speaking, Scheme programs will run on XLisp—the exceptions involving use of special features like arbitrary-size integers (known in lisp parlance as "bignums"). Accordingly, any of the several good textbooks on Scheme [see the References] will prove useful to the XLisp programmer. Scheme is not an "obscure" language—it is the language of choice in Computer Science departments at many universities, including MIT where it was developed in 1975.

This tutorial is intended as a briefest-possible introduction for the C programmer who is not assumed to have any background in lisp programming.

1.1 XLisp Data Types

1.1.1 Numeric data types

XLisp's numeric data types are a subset of C's:

integer	corresponds to C's long int
number	corresponds to C's double

Table 1

(Full Scheme also has bignums, rationals, and complex numbers.)

1.1.2 Non-numeric data types

XLisp also has the following non-numeric data types:

string	identical to C's, e.g. "Hello, world!"
character	represents the ASCII set. The notation is `#\a` for what C would write as 'a'. Special cases are `#\space` and `#\newline`.
vector	An array of objects of inhomogeneous type, for example `#(27 "George" #\z).` Roughly equivalent in C to an array of void pointers.
Boolean	`#t` or `#f`, similar to C's 1 and 0. False (`#f`) is also represented, for traditional reasons, by the empty list `()` and equivalently by the symbol `NIL`. In C, anything non-zero is true; the same thing happens in XLisp, where anything different from `#f` (or the equivalents, `()` or `NIL`) is treated as `#t`.
symbol	In C, symbols are used as identifiers, but symbols are not a data type. In XLisp, where symbols are also used as identifiers, the value of a symbol *can* be another symbol.
procedure	XLisp allows run-time creation of procedures, which are first class objects (i.e., can be passed as arguments, returned as values, or created anonymously in expressions).
Class, Object	XLisp provides a Smalltalk-like Object System.

Table 2

These are the "atomic" data types of XLisp. XLisp uses these to build up general Symbolic Expressions, called S-Expressions. The prototypical S-Expression is the `list`, written as a space-separated list of elements enclosed in parentheses:

```
(a1 a2 ... aN)
```

The list elements can be arbitrary data objects including other lists.

2. THE XLISP EVALUATOR

XLisp is an interpreter which normally runs in a READ-EVAL-PRINT loop: that is, it repeatedly READs an S-Expression typed at the keyboard (or input from a file), EVALuates the S-Expression, and PRINTs the value on the console.

Constants evaluate to themselves—for example:

```
[1] 27
27
[2] "George"
"George"
[3] _
```

(Notice that XLisp numbers its inputs, allowing them and their associated responses to be subsequently recalled, as described below.)

Symbols that have been assigned a value will return that value when evaluated:

```
[3] nil
()
[4] +
#<Subr +>
[5] _
```

The symbol `NIL` is "bound" to the empty list, `()`, which is also the value of the Boolean `#f`. The symbol + is bound to (has as its value) a pre-defined subroutine which adds numbers.

Lists are similar to function calls, in which the first list element names the function and the remaining elements are arguments. For example,

```
[5] (+ 3 4 10)
17
[6] (string-append "Fred" "+" "Ginger")
"Fred+Ginger"
[7] _
```

In each case, the first list element is a procedure; the Evaluator proceeds to evaluate each of the remaining list elements and use their values as arguments to that procedure. This is a recursive definition, because list elements may themselves be lists that need to be evaluated:

```
[7] (+ (* 3 5) 100)
115
[8] _
```

Notice here that the second list element—`(* 3 5)`—is a list which evaluates to the product of 3 and 5, i.e. 15. The third list element, 100, is a constant that evaluates to itself. The first list element, whose value is the addition procedure, adds the values of the other list elements to give 15 + 100 = 115.

As a final example, we mention the functions alluded to above for recalling expressions previously entered, and the values returned. `%e` takes a single integer argument, and returns the S-expression typed as input; the value returned can be recalled using `%v` in the same way:

```
[8] (%e 5)
(+ 3 4 10)
[9] (%v 6)
"Fred+Ginger"
[10] _
```

2.1 Special Forms

List-evaluation typically involves evaluating all list elements, followed by applying the value of the first element—which must be a procedure—to the values of the remaining elements, which are arguments to that procedure. There are certain situations where the Evaluator must deviate from this practice, and not *automatically* evaluate each list element before making the procedure call. These lists are known as *special forms*, and are distinguished by special pre-defined symbols which appear as the first list element. For example, the list `(define x 100)` is recognized as a special form because the Evaluator knows about the reserved symbol `define`. Another such reserved symbol is `if`—both are described below. (Informally, one sometimes speaks of the reserved symbol itself as the special form.)

2.1.1 DEFINE

`Define` is used to introduce symbols into symbol-tables, and optionally to assign values to those symbols. For example:

```
[8] (define x 100)
x
[9] x
100
[10] (+ x 1)
101
[11] _
```

Statement [8] creates a symbol-table entry for `x`, and binds it to the initial value 100. Typing `x` at XLisp in [9] shows that it was paying attention. Statement [10] shows that `x` can be used in an expression, and its value will be looked up and used.

`Define` cannot be an ordinary procedure, for then its arguments would have to be evaluated and we know that its first argument, `x`, is not guaranteed to have a value. In fact, it may be unknown to the system (that is, not found in the symbol-table, which is different from having an entry but simply not having been assigned a value). In fact, the whole purpose of `define` is to introduce a new symbol and assign it a value. So the Evaluator recognizes `define` and handles the situation specially.

2.1.2 IF

`If` has a similar problem to deal with: its general form is

```
(if test consequent alternative)
```

and it may be important *not* to evaluate *both* the consequent and the alternative. This situation arises in C:

```
c = getc(in);
if (c != EOF)
    putc(c, out);              // do this--
else
    fclose(out);               // or this, but not both!
```

In lisp, the corresponding code looks like:

```
(set! c (read-char in))         ; set!, like define, is a special form
(if (not (eof-object? c))       ; if we're not at end-of-file
    (write-char c out)          ; then write the char to output
    (close-port out))           ; else, close the output port
```

With ordinary procedures, the Evaluator evaluates *all* the list elements before applying their values as arguments. Therefore, `if` is treated as a special form: the first argument, `test`, is evaluated, and depending on whether it returns true or false *either* the second *or* the third argument will be evaluated, but never both. The value of the one chosen is the value of the `if` expression:

```
[11] (if (zero? x) "Can't take reciprocal of Zero!" (/ 1 x))
0.01
[12] _
```

Here the `test` part is an ordinary procedure call: the function `zero?` returns `#t` only if its argument is in fact 0. If that is the case, the `if` will evaluate the second argument, which is a string constant, and return that—it *won't* attempt to evaluate the third argument which would involve the forbidden deed. In our case, `x` is 100, so the `test` returns `()`—i.e., `NIL` which is the same as `#f`—and the reciprocal is computed by using the division procedure / to divide 1 by `x`.

Generally, we can treat evaluation of a list by a single rule: the first element specifies some type of operator and the remaining elements are its operands. Special forms, in which the first element is one of a few predefined keywords, are handled specially—some of the arguments being evaluated and others not, as dictated by the keyword. (The user can augment this keyword set using the macro facility, but this is a separate topic.) Other than this, all list elements are evaluated, and the first had better turn out to be a procedure that can accept the values of the remaining elements as its arguments.

2.2 Nested Environments and LET

An Environment is a symbol-table. When we type S-Expressions at the XLisp Evaluator, symbols occurring in these expressions are looked up in the global environment. For example, when we type

```
[1] (+ 3 5)
8
[2] _
```

the symbol + is looked up; its value, in the global symbol-table, is a procedure that adds numbers. If a symbol occurring in the expression cannot be found, an error will occur. We have seen above how `define` is used to introduce a new symbol into the current environment and assign it an initial value.

Notice the use of the modifier *current* in referring to the environment: environments can be nested, and a symbol that occurs in a nested environment can "shadow" the value of that symbol defined in an outer environment (including the global environment). This is the same block structuring that is seen in C:

```
int x = 85;
{
    int x = -2;
    printf("inner x = %d\n", x); // prints "x = -2"
}
printf("outer x = %d\n", x);     // prints "x = 85"
```

Nested environments are created explicitly with the special form `let`:

```
[2] (define x 85)
x
[3] (let ()
      (define x -2)
      (print x))
-2
#t
[4] x
85
[5] _
```

Here at the "top level," we introduced the symbol **x** into the global environment and gave it the value 85. In statement [3], we created a **let** form: **let**'s first argument is a list of symbols being created in a new symbol-table (environment) that is linked to the global environment. Here the list is empty: no new symbols are being introduced. That is done by the **define**, which is invoked within the scope of the **let**. This **define** introduces the symbol **x** and assigns it the value -2. When the **print** function is invoked in the next line, the Evaluator looks up **x** in the innermost enclosing environment, which is the one created by the **let**: there it finds the **x** that was deposited by the **define**, and discovers that its value is -2. The effect of invoking the **print** procedure is to print the value of **x** (here -2). (The **#t** on the line below the -2 is the value *returned* by the **let** expression: the READ-EVAL-PRINT loop prints the value of the **let** expression it has just EVALuated.)

Statement [4] shows that the global value of **x** has not been affected. The nested environment went "out of scope" when we crossed the second parenthesis after the **print**—the one that balanced the open parenthesis of the **let** itself.

The normal syntax of **let** is

```
(let ((sym1 val1) (sym2 val2) ...)
  exp1 exp2 ...)
```

where **sym1, sym2, ...** are symbols that are being introduced into the nested environment, and **val1, val2, ...** are arbitrary expressions whose computed values are assigned to the associated symbols. The body of the **let** consists of a series of S-Expressions **exp1, exp2, ...** which are evaluated sequentially inside the nested environment. Thus, the symbols **sym1, ...** can occur in these expressions and will have the values that have been locally assigned. The value returned by the **let** is the value of the final expression found in its body.

The example shown above in [3] could be reworked equivalently as follows:

```
[5] (let ((x -2))
      (print x))
-2
#t
[6] x
85
[7] _
```

Again we see that the global value of **x** is unchanged.

It is important to understand that that the expressions **val1, ...** in the general **let** form above are evaluated in the innermost surrounding environment that encloses the **let**. For example:

```
[7] (let ((x -2)
          (y (+ 1 x)))
      (+ x y))
84
[8] _
```

The value of the **let** is the value of the single expression in its body: **(+ x y)**. Here, the evaluator searches for the value of the symbol + and finds it in the global environment with its usual value (the addition procedure). **x** is looked up in the same manner, but here a value is found in the innermost

enclosing environment: -2. **y** is found in this same environment and has the value 86. That arises because the expression **(+ 1 x)**, whose value is assigned to **y**, is evaluated in the environment that encloses the **let**, and in this environment **x** has the value 85. Consider instead the expression:

```
[8] (let ((x -2))
      (let ((y (+ x 1)))
        (+ x y)))
-3
[9] _
```

Now we have a pair of nested **let**s: when the symbol **y** is introduced in the second of these, the **x** comes from the previous **let** which shadows the **x** in the global environment. In evaluating **(+ x y)** , the Evaluator searches back along the chain of enclosing environments until each of the symbols is found. **y** is found in the innermost enclosing environment, and **x** is located in the environment that immediately surrounds that.

The special form **let*** allows a simple abbreviation for nested **let**s:

```
(let ((s1 v1))
  (let ((s2 v2))
    (let ((s3 v3))
      exp1
      exp2)))
```

can be written more succinctly with **let***:

```
(let* ((s1 v1)
       (s2 v2)
       (s3 v3))
  exp1
  exp2)
```

Thus the example of [8] could be rewritten as:

```
[9] (let* ((x -2)
           (y (+ x 1)))
      (+ x y))
-3
[10] _
```

2.3 SET!

C programmers normally code their algorithms as a series of assignment statements: an expression is computed and the value is assigned to a variable, as in

```
x = a + b;
```

The XLisp-equivalent expression is

```
(set! x (+ a b))
```

SET!, like DEFINE, is a special form that treats the first argument — here **x** — as a target symbol to receive the value computed from the second argument. Normally, DEFINE is used to introduce new symbols into the current environment, and SET! is used to alter (reassign) the values bound to these symbols.

In formal Scheme, it is an error to use SET! on a new symbol, or to re-DEFINE an existing one; most conversational Schemes, including XLisp, are more tolerant, and will allow DEFINE and SET! to be used interchangebly at the top level. But in nested environments, the behavior is very different: SET! will search for the target symbol starting in the environment active at its point of invocation, looking backward through the nested hierarchy, and modify the first instance of that symbol it finds. If the

symbol cannot be found at all, it will be introduced and assigned at the top level — that is, in the global environment. DEFINE, on the other hand, only acts within its local environment. For example:

```
[1] (define x 85)
x
[2] (define y -10)
y
[3] (let ((y 100))
      (set! x 7)
      (set! y 5)
      (format #t "x = ~A, y = ~A~%" x y))
x = 7, y = 5
()
[4] x
7
[5] y
-10
[6] _
```

In statements [1] and [2], we introduce global variables **x** and **y**, assigning them the values 85 and -10, respectively. In [3], we create a nested environment in which a local symbol **y** occurs with a value of 100. The two SET! statements reassign **x** and **y** to 7 and 5, respectively. The FORMAT statement — described later — prints the values of **x** and **y** and shows that they have changed. But notice from [4] and [5] that the global value of **x** has been altered, while the top-level **y** remains unmolested. This illustrates the behavior of SET! outlined above. Had DEFINE been used in place of SET!, the printed output would have been the same, but the global values of both **x** and **y** would have been unaffected.

2.4 Quoting

Symbols are a data type, but how could you assign a symbolic value to a variable? For example, I want to create a variable **x** and initialize it to have the symbolic value **done**. The following attempt fails:

```
[1] (define x done)
error: unbound variable - done
```

and the interpreter enters a Debug loop from which we can escape by entering the function call (**reset**).

The problem is that the **define** wants to assign to **x** the *value* of **done**, and so looks up **done** in its symbol table and doesn't find it. We're not getting what we intended here: we don't want the value associated with the symbol **done** but the symbol **done** itself. There is a special form, **quote**, for dealing with just this situation:

```
[1] (define x (quote done))
x
[2] x
done
[3] _
```

When the Evaluator sees a list whose first element is the keyword **quote**, it takes the second list element *as written* and passes it along as the value of the **quote**. Here's a more interesting example:

```
[3] (define y (quote (+ 1 2)))
y
[4] y
(+ 1 2)
[5] _
```

Now the variable **y** has been assigned, not just a symbol, but a symbolic expression in the form of a list. Without the **quote**, **y** would have been assigned the *value* of this list, namely 3.

Quoting is so common that the function **read** , which parses input before handing it over to EVAL, recognizes a convenient shorthand: a single quote ' placed in front of an expression quotes the expression. That is, **'done** is treated the same as (**quote done**), and the statement [3] above could equally have been entered:

```
[3] (define y '(+ 1 2))
```

with exactly the same effect.

Quoting in XLisp is similar to quoting in English, as illustrated by the neat example I read in a book long ago:

Chicago is a windy city; "Chicago" is a seven-letter word.

3. LAMBDA

Procedures are created using the keyword **lambda**—for example:

```
[1] (define square (lambda (x) (* x x)))
square
[2] _
```

Here the symbol **square** is introduced into the global environment with a value resulting from the expression **(lambda (x) (* x x))**. The keyword **lambda** tells the Evaluator that a procedure is being created. Immediately following the **lambda** is the formal argument list: in this case, there is a single argument represented by the symbol **x**. Following the argument list is a sequence of one or more expressions to be evaluated when the function is invoked: the value of the last of these expressions will be the value the procedure returns.

The procedure **square** can now be invoked the same as any built-in XLisp function:

```
[2] (square 10)
100
[3] _
```

Here's how the Evaluator arrives at this result. It notices that **square** is defined as a procedure, with a single formal argument **x**. It creates a nested environment in which the formal argument, **x**, is bound to (i.e., assigned) the value of the actual argument 10. Then the body of the procedure is evaluated in this environment. Here the body consists of the single expression **(* x x)**. **x** might have a global value, but it is the local value that is used; the value 100 results from the multiplication, and is returned as the value of **(square 10)**.

Procedures capture the local environment in which they are created; this allows them to hold local state. Here is a procedure of no arguments that simply returns the number of times it's been called:

```
[3] (define how-many-times
      (let ((count 0))
        (lambda ()
          (set! count (+ count 1))
          count)))
how-many-times
[4] (how-many-times)
1
[5] (how-many-times)
2
[6] _
```

The symbol **how-many-times** is assigned the value of the **lambda** expression, which is a function of no arguments that increments the local variable **count** and then returns the incremented value. **count** is defined in a local environment seen only by the created procedure; no other occurrence of **count** can possibly refer to the same datum:

```
[6] (define count 100)
count
[7] (how-many-times)
3
[8] count
100
[9] _
```

The mechanism here is similar to the use of static locals in a C function, to retain state across invocations.

Procedures are first-class data objects that can be passed as arguments and returned as values. As a simple illustration, we define a procedure `one-upper` that takes a numerical function like `square` and returns a "one-upping" function that always returns a value 1 larger.

```
[9] (define one-upper (lambda (f)
                        (lambda (x)
                          (+ (f x) 1))))
one-upper
[10] _
```

Here `one-upper` is defined as a procedure taking a single argument `f` which represents a numerical function. The value returned by `one-upper` is the value of the `lambda` expression `(lambda (x) ...)`, that is, a function of a single argument `x` that computes a value. The value it computes is given by the expression `(+ (f x) 1)`, which is the sum of 1 and the result of applying the given function `f` to `x`. We can use `one-upper` to transform our `square` procedure into a procedure that computes 1 more than the square of its argument:

```
[10] (define sq++ (one-upper square))
sq++
[11] (sq++ 10)
101
[12] _
```

If the newly-created one-upping procedure is going to be used only once, there is no need to assign it to a variable with `define`: it can be created anonymously and used in place:

```
[12] ((one-upper sqrt) 64)
9
[13] _
```

Here we see an illustration of the recursive nature of list evaluation: the first list element is itself a procedure call that evaluates to a procedure (namely, the one-upper of the square root procedure); that procedure is then applied to the value of the second element, which is the constant 64.

A more complicated example of transforming one procedure into another is the following, which I've had occasion to use in Numerical Analysis work. We define a procedure `make-counting-version` that takes, as its argument, any function like `square` or `sqrt` that computes numerical values, and returns a new version of that function which keeps track of the number of times it's been called:

```
[13] (define make-counting-version
       (lambda (function)
         (let ((count 0))
           (lambda (x)
             (cond ((number? x) (set! count (+ count 1))
                                (function x))
                   ((eq? x 'count) count)
                   ((eq? x 'reset!) (set! count 0))
                   (else 'huh?))))))
make-counting-version
[14] _
```

The special form `cond` is a powerful conditional that presents a number of "phrases", each as a list. In each phrase, the first element is evaluated—if the result is false, then we proceed immediately to the next phrase. When a phrase is encountered whose first element is true (the final `else` always counts as true), then the remaining expressions in that phrase are evaluated sequentially and the value of the final expression is immediately returned as the value of the `cond`.

Thus, `make-counting-version` is a procedure of a single argument, `function`, which is presumed to take a numerical argument and return a numerical value. As in the example of `how-many-times` above, we create a local counter for use of the procedure we return—that procedure being created by the `(lambda (x) ...)` appearing inside the nested environment of `(let ((count 0)) ...)`.

The function we return takes its argument **x** and first asks if **x** is a number. If it is, then we increment the counter and return the value that **function** would have returned. But we allow the caller to supply non-numerical values: the symbolic argument **'count** prompts us simply to return the counter value, and the symbolic argument **'reset!** resets the counter. Any other type of argument is unexpected and provokes the response **'huh?**.

```
[14] (define csq (make-counting-version square))
csq
[15] (csq 4)
16
[16] (csq -9)
81
[17] (csq 'count)
2
[18] (csq 'count)
2
[19] (csq 5)
25
[20] (csq 'count)
3
[21] (csq 'reset)
huh?
[22] (csq 'reset!)
0
[23] (csq 'count)
0
[24] _
```

3.1 Creation and manipulation of Lists

The simplest way to create a list is with the procedure **list**, that takes any number of arguments and returns a list of those arguments:

```
[1] (define example-list (list 'a 100 "George"))
example-list
[2] example-list
(a 100 "George")
[3] _
```

XLisp has built-in functions **first**, **second**, **third**, and **fourth** for extracting the early elements of any list:

```
[3] (second example-list)
100
[4] _
```

For historical reasons, the name **car** is often used in place of **first**:

```
[4] (car example-list)
a
[5] _
```

The nth element of a list (0-based indexing) can be extracted using the function **list-ref**:

```
[5] (list-ref example-list 2)
"George"
[6] _
```

One can extract the remainder of a list after removing the first element by using the function **rest**:

```
[7] (rest example-list)
(100 "George")
[8] (rest (rest example-list))
("George")
[9] _
```

For historical reasons, **rest** is commonly known by the synonym **cdr**—pronounced "could'-er"). Many recursive procedures involve "CDRing" down a list, as for example the following which counts the number of elements in a list:

```
[9] (define length-of-list
      (lambda (L)
        (print L)
        (if (null? L)
          0
          (+ 1 (length-of-list (rest L))))))
length-of-list
[10] _
```

This just says that if the given list, **L**, is empty (that is, a **null** list), then the length is 0; else, the length is 1 more than the length of the **rest** of the list. We've inserted the (**print L**) statement just so we can see how this works when we call it:

```
[10] (length-of-list example-list)
(a 100 "George")
(100 "George")
("George")
()
3
[11] _
```

The function **length-of-list** calls itself recursively, each time on the whittled-down sublist that remains after removing the first element. The printed output shows this. Finally the list is reduced to **NIL**—that is, the empty list **()**—and the procedure simply returns 0. This returned value is handed back up the ladder to previous invocations that add 1 to it, and the final result—3—is printed by the READ-EVAL-PRINT loop in the usual way.

The built-in function **length** returns the length of any list, without printing out its argument as above.

3.1.1 Building up lists with CONS

One can use the procedure **cons** to add elements onto the front of a list:

```
[11] (cons 'foobar example-list)
(foobar a 100 "George")
[12] _
```

Using **car**, **cdr**, and **cons** together, we can write our own version of the standard function **reverse** that takes a list and makes a new list with the same elements in reverse order. It's convenient to start with a "helper" function that takes two lists as its arguments, and peels elements off the front of the first list while pasting them onto the front of the second:

```
[12] (define helper
       (lambda (old-list new-list)
         (if (null? old-list)
           new-list
           (let ((element (car old-list)))
             (helper (cdr old-list) (cons element new-list))))))
helper
[13] _
```

Notice the recursive action of **helper**: if **old-list** is not empty, it extracts its first *element* using **car** (or **first**), and then calls itself with **old-list** replaced by the **cdr** (**rest**) of its previous value, and **new-list** replaced by its previous value with *element* CONSed onto the front of it. All we have to do now is start up the target list initially empty, i.e., with **old-list** initialized to **NIL**:

```
[13] (define list-reversed
       (lambda (L)
         (helper L nil)))
list-reversed
[14] (list-reversed example-list)
("George" 100 a)
[15] _
```

3.1.2 Concatenating lists

The built-in procedure **append** takes any number of argument lists and returns their "concatenation":

```
[15] (append '(a b c) '(1 2 3 4 5) '("Harry" "Fred"))
(a b c 1 2 3 4 5 "Harry" "Fred")
[16] _
```

None of the functions described so far alters the argument lists: new lists are always created and returned. There are so-called "destructive" list functions that do modify their arguments; their names end in the exclamation mark ! as a warning. For example, the destructive version of **append** is **append!**—it actually splices the lists in place.

3.2 Procedure definition in XLisp

The special form **define** allows us to replace a procedure definition like

```
[1] (define square (lambda (x) (* x x)))
square
[2] _
```

with the following equivalent construction:

```
[2] (define (square x) (* x x))
square
[3] _
```

define, in looking at its first argument, sees not a symbol but rather the expression **(square x)**, which has the appearance of a procedure call. Being intelligent, **define** infers that a procedure named **square** is being defined, and supplies the **lambda** implicitly.

This process is analogous to the defining mechanism used in C, where a variable is declared by using it in an expression that evaluates to a known type. For example,

```
int (*Func)(double, int);
```

declares **Func** to be a pointer to a function taking a **double** and an **int** as arguments and returning an **int** as value. Likewise,

```
[3] (define ((one-upper f) x)
      (+ (f x) 1))
one-upper
[4] _
```

declares `one-upper` to be a function that takes an argument `f` and returns a function that takes an argument `x`. Nothing here constrains the data type of `f` or `x`, but the procedure body—`(+ (f x) 1)`—won't work if `f` isn't a function and `x` an argument it can act upon.

On examining the definition in [3], you may come to find it more transparent than the equivalent definition given earlier in terms of `lambda` expressions:

```
(define one-upper (lambda (f)
                    (lambda (x)
                      (+ (f x) 1))))
```

The `lambda` form is actually more flexible, because it allows us to insert local environments in front of it, as in the earlier example

```
(define how-many-times
          (let ((count 0))
            (lambda ()
              (set! count (+ count 1))
              count)))
```

Trying to do the same thing without explicitly using `lambda` is tricky; for example

```
(define (how-many-times)
  (let ((count 0))
    (set! count (+ count 1))
    count))
```

recreates the local variable `count` each time the procedure is entered; the return value is always 1. The better attempt:

```
(let ((count 0))
  (define (how-many-times)
    (set! count (+ count 1))
    count))
```

fails because the procedure `how-many-times` is embedded in the same nested environment as `count`, and disappears as soon as we exit the `let`. But the following works:

```
[4] (define how-many-times)
how-many-times
[5] (let ((count 0))
      (define (foo)
        (set! count (+ count 1))
        count)
      (set! how-many-times foo))
#<Procedure FOO>
[6] _
```

Statement [4] introduces the symbol `how-many-times` into the global environment, without bothering to give it a value. (XLisp gives it the default value `NIL`; other lisps mark it explicitly as unassigned or "unbound". I used to enjoy one such interpreter that would respond to my using the undefined symbol `FRANKENSTEIN` with the error message "`FRANKENSTEIN UNBOUND`".)

Statement [5] proceeds as above to define a local function—now called `foo`—that lives in a private environment with its local data `count`. `foo`, like `count`, will go out of scope as soon as we exit the `let`—hence, before exiting, we assign the value stored in `foo` to the globally visible `how-many-times`. The value assigned using the `set!` is what XLisp prints as the value of the `let`.

Both styles of procedure definition—with and without the explicit **lambda** —have extensions to allow optional arguments and a variable number of arguments. XLisp borrows additional syntactical extensions from Common Lisp that make these features especially convenient, and these are described and illustrated now.

3.2.1 Optional, rest, and keyword arguments

Optional arguments can be specified using **&optional**. For example,

```
[6] (define (foo a b &optional (c 17))
      (list a b c))
foo
[7] _
```

defines **foo** as a function taking two required arguments—**a** and **b**—and an optional argument **c** that has a default value of 17. Since **foo** simply returns its arguments in a list, it is easy to see by experiment what those arguments are:

```
[7] (foo 10 20)
(10 20 17)
[8] (foo 10 11 12)
(10 11 12)
[9] _
```

Any number of arguments can follow **&optional**; here's an example using **lambda** explicitly:

```
[9] (define bar (lambda (x &optional (y 10) z (w "yes"))
                  (list x y z w)))
bar
[10] _
```

Here **bar** has a required argument, **x**, and three optional arguments, **y**, **z**, and **w**. **y** and **w** have specified defaults of 10 and "yes" respectively; no default has been stipulated for **z** which accordingly defaults to **()**—i.e., **NIL**. Thus:

```
[10] (bar 1)
(1 10 () "yes")
[11] (bar 1 2 3)
(1 2 3 "yes")
[12] _
```

Procedures with a variable number of arguments can be defined using **&rest**, as in the example:

```
[12] (define (foobar x y &rest z)
       (list x y z))
foobar
[13] _
```

The meaning here is that **foobar** may be invoked with any number of arguments greater than or equal to 2, since the first two arguments—**x** and **y**—are required. If there are more than two arguments when **foobar** is called, all the "extra" ones will be collected into a list and that list will be bound to the **&rest** argument, **z**. Thus:

```
[13] (foobar 1 2)
(1 2 ())
[14] (foobar 1 2 3)
(1 2 (3))
[15] (foobar 1 2 3 4 5)
(1 2 (3 4 5))
[16] _
```

The built-in procedure **list** could be succinctly redefined using **&rest** as **(lambda (&rest L) L)**.

&optional arguments can coexist with the single **&rest** argument, as long as the latter comes at the end of the argument list.

Another very useful extension allows non-positional keyword arguments, which are prefaced in the argument list by **&key**. Consider the following example:

```
[16] (define (page &key (color 'white) (width 8.5) (height 11))
       (list color width height))
page
[17] _
```

Defaults are specified as with **&optional**; likewise here a value of **NIL** is used if no default is given. But here we can override the defaults in an order-independent way, by using the **&key** variable name preceded by a colon as in these examples:

```
[17] (page)
(white 8.5 11)
[18] (page :height 17)
(white 8.5 17)
[19] (page :width 4 :color 'grey)
(grey 4 11)
[20] _
```

This arrangement is extremely convenient for functions that take many arguments having natural default values. Having to remember the order of numerous arguments is a difficulty we hereby avoid; and only the arguments that vary from their defaults need be specified at all. The use of the keyword makes the invocation self-documenting.

&key can be used in conjunction with **&optional** provided **&key** comes last. **&key** cannot be used with **&rest**, each needing to come last if used at all.

4. XLISP FILE I/O

XLisp's facilities for reading and writing files are based on C's: one can work with text or binary files; in the latter case, byte order can be controlled.

We'll focus on output in what follows, and begin with standard printing functions **`display`**, **`write`**, and **`format`**.

`Display` and **`write`** each take a single required argument—the value to be printed—and an optional argument which is the output port to which the printing is done. This output port, if omitted, defaults to *STANDARD-OUTPUT*:

```
[1] (display "Hello")
Hello
#t
[2] (write "Hello")
"Hello"
#t
[3] _
```

In each case, the first line of output results from the action of the procedure called; the second line (**`#t`**) is the value returned by the procedure and printed by the READ-EVAL-PRINT loop in the standard way.

Notice that **`write`** outputs the string with the double-quotes. Generally, **`display`** is the right choice for output that people will read; **`write`** attempts to structure its output so that the object printed could be reconstructed if read back in using **`read`**.

XLisp's **`format`** is a much-simplified version of the Common Lisp procedure that serves the purpose of C's printf. The first, and required, argument is either **`#t`**, **`#f`**, or an output port. **`#t`** is used to have the same effect as specifying *STANDARD-OUTPUT*. **`#f`** means that printing will be done to a (newly-allocated) string, which is returned as the value of the **`format`** call.

The second, also required, argument is the format string itself, analogous to **`printf`**'s format string. The special formatting character, serving the same purpose as **`printf`**'s %, is the tilde ~. XLisp's **`format`** recognizes these constructs:

Argument	Definition
~A	print the next argument as with DISPLAY
~S	print the next argument as with WRITE
~X	print the integer argument in hex, using 8 hex digits
~%	print a new line
~&	print a new line if not at the beginning of a line
~~	print a ~ (tilde)

Table 3

For example:

```
[3] (define greeting "Howdy!")
greeting
[4] (format #t "using DISPLAY, it's ~A~%using WRITE, it's ~S"
               greeting greeting)
using DISPLAY, it's Howdy!
using WRITE, it's "Howdy!"
()
[5] _
```

The final `NIL` is the value returned by `format`.

File I/O is adequately described in that section of the XLisp Manual: the same flexibility offered by C is in evidence. For example:

```
[5] (define p (open-output-file "foo.txt"))
p
[6] p
#<File-stream #x1d212e0:476a10>
[7] (port? p)
#t
[8] _
```

In [5] we open a text output file named "foo.txt". [6] shows that the value returned by the call is a File-stream—something like a file pointer in C. [7] shows that `p` is of type `port`. The following sequence can now be used to write to the newly-created file:

```
[8] (display "Here's a sentence!" p)
#t
[9] (newline p)
#t
[10] (format p "-1 in hex is ~X~%" -1)
()
[11] (close-port p)
()
[12] _
```

If we now examine the file "foo.txt", we discover that it contains these lines:

```
Here's a sentence!
-1 in hex is ffffffff
```

The same effect could have been achieved using the procedure (`call-with-output-file str proc`)—here `str` is a string that names the output file to create, and `proc` is a procedure taking one argument (which will be a port):

```
[12] (call-with-output-file
       "foo.txt"
       (lambda (p)
         (display "Here's a sentence!" p)
         (newline p)
         (format p "-1 in hex is ~X~%" -1)))
#t
[13] _
```

Notice that the `proc` argument is created explicitly in place using `lambda`; the single argument `p` is just the port which results from opening the file. What's nice here is that the port is closed automatically when `call-with-output-file` completes.

There doesn't happen to be a `call-with-append-file`, which might be convenient; here's how one could be written to work in the manner shown above:

```
(define (call-with-append-file str proc)
  (let ((p (open-append-file str)))
    (if (not (port? p))
      'error-opening-file!
      (let ((result (proc p)))
        (close-port p)
        result))))
```

There is another kind of output stream that is sometimes useful: instead of being associated with an output file, it references an in-memory extensible string. See the XLisp Manual under "String Stream Functions". The call

```
(define s (make-string-output-stream))
```

creates a string-output-stream and assigns it to the symbol **`s`**. Now **`s`** can be used as an output port, and will receive whatever is written to it. The call

```
(define str (get-output-stream-string s))
```

extracts the output-stream string from **`s`** and binds it to **`str`**.

This is a little like using **`sprintf()`** in C to write formatted output to a character array, a notable difference being that in C the array doesn't just grow dynamically to accommodate its input!

5. XLISP OBJECT SYSTEM

XLisp provides a single-inheritance Object System inspired by Smalltalk. We'll illustrate here a few of its features; more details can be found in the XLisp Manual.

Here's a definition of a class of two-dimensional geometric points:

```
[1] (define-class point
      (instance-variables x y))
#<Class:point #x1618c94>
[2] _
```

(The return value is a printed-representation of the `point` class; the hex number (prefaced in XLisp with `#x`) is an internal address.)

This lets us create `point` objects by sending the `point` class the message '`new`:

```
[2] (define p1 (point 'new))
p1
[3] _
```

Generally, sending a message to an object (or a class) is written in the same syntax as a procedure call, with the object (or class) occupying the procedure position (first list element) and the first argument being the message which is typically a quoted symbol. The message '`new`, sent to a class, always creates a new object instance belonging to that class. Here we've created a new `point` object and bound it to the symbol `p1`.

All objects respond to the '`show` message by displaying their internals (I give an abbreviated version below):

```
[3] (p1 'show)
Instance variables:
  x = ()
  y = ()
[4] _
```

We need to write new "methods" to allow setting of the instance variables. For example:

```
[4] (define-method (point 'set-x! value)
      (set! x value))
set-x!
[5] (define-method (point 'set-y! value)
      (set! y value))
set-y!
[6] _
```

Notice that inside the body of a method definition, instance variables can be referred to by name.

Now we can establish coordinates for our point:

```
[6] (p1 'set-x! 3)
3
[7] (p1 'set-y! 4)
4
[8] (p1 'show)
Instance variables:
  x = 3
  y = 4
[9] _
```

Other methods can be added as required: for example, one to compute the distance of a point from the origin (0, 0) according to the formula sqrt(x^2 + y^2):

```
[9] (define-method (point 'distance-from-origin)
      (let ((xsq (* x x))
            (ysq (* y y)))
        (sqrt (+ xsq ysq))))
distance-from-origin
[10] _
```

The distance of `p1` from the origin = sqrt(3^2 + 4^2) = sqrt(9 + 16) = sqrt(25) = 5:

```
[10] (p1 'distance-from-origin)
5
[11] _
```

Note from statement [3] above that a newly created point doesn't have numerical coordinates (the instance variables are initialized to `NIL`); this would cause the `'distance-from-origin` method to fail if applied to a new point. There is a standard way to customize the initialization of new objects: override the default `'initialize` method as illustrated here:

```
[11] (define-method (point 'initialize &optional (xval 0) (yval 0))
       (set! x xval)
       (set! y yval)
       self)
initialize
[12] _
```

The symbol `'self` is bound, within the body of an object method, to the object itself; the `'initialize` method must always return the initialized object as value, hence the final line after the `set!` statements. Now we can pass to the `class` method '`new` the same arguments we want it to pass to the `'initialize` method for the newly created instance:

```
[12] (define p2 (point 'new -1 2))
p2
[13] (p2 'show)
Instance-variables:
  x = -1
  y = 2
[14] _
```

Since we wrote our `'initialize` method with optional arguments, omitting explicit initializing data is safe:

```
[14] (define p3 (point 'new))
p3
[15] (p3 'show)
Instance-variables:
  x = 0
  y = 0
[16] (p3 'distance-from-origin)
0
[17] _
```

This brief introduction has made no mention of class variables or inheritance (other than the use of methods '`new` and '`show` which are inherited from superclasses). Our goal was simply to illustrate the syntax of class and object creation, method definition, and message passing.

6. NAMED LET

The general form of **let** given in the XLisp Manual—under "Binding Forms"—is:

```
(let [name] bindings body)
```

where **bindings** has the form (**(variable init)** ...), and **body** is any sequence of S-Expressions to be evaluated sequentially, the value of the last being the value returned by the **let**.

Although we haven't mentioned it up till now, the **let** can be **named** by placing an optional symbol between the **let** and the **bindings**. To explain the meaning and use of this construct, we begin by noting that **let** is equivalent to invocation of an anonymous procedure. Consider the following example:

```
[1] (let ((x 5) (y 10))
      (+ x y))
15
[2] _
```

The body of the **let**—here the single S-Expression **(+ x y)** — is evaluated in a local environment with symbols **x** and **y**, in which **x** has been bound to 5 and **y** to 10. The result of the evaluation is 15.

The expression in [1] is exactly equivalent to the following:

```
[2] ((lambda (x y) (+ x y)) 5 10)
15
[3] _
```

Notice that in [2] we have a list whose first term is the **lambda** expression **(lambda (x y) (+ x y))**. Evaluating this expression produces a procedure that is then applied to the arguments 5 and 10. How is the procedure applied to the arguments? First, a local environment is created containing the formal arguments **x** and **y**. Then the formal arguments are bound to the actual arguments—5 and 10—and then the body of the **lambda** expression—**(+ x y)**—is evaluated in this environment. This is **exactly** the action of the **let** in [1]!

Thus, **let** creates a procedure whose formal arguments are the VARIABLEs in its **bindings** list. The actual arguments are the values of the INIT expressions in the **bindings** list. Applying the procedure to the arguments means evaluating the **body** in this environment.

The procedure, as in [2] above, is anonymous (that is, unnamed). This means there's no way for the implied procedure to call itself recursively. The point of the **named-let** construction is to give the implicitly-defined procedure a name so that it may refer to itself.

For example:

```
[3] (let loop ((i 1))
      (if (> i 3)
        'done
        (begin
          (print i)
          (loop (+ i 1)))))
1
2
3
done
[4] _
```

Here the procedure created by the **let** is named **loop**. It has a formal argument, **i**, and the body returns '**done** if **i** is greater than 3. If **i** is not greater than 3, **i** is printed and **loop** is called recursively with **i** incremented by 1. This is equivalent to the following:

```
[4] (let (loop)
      (set! loop (lambda (i)
                   (if (> i 3)
                     'done
                     (begin
                       (print i)
                       (loop (+ i 1))))))
      (loop 1))
1
2
3
done
[5] _
```

Here the `let` creates a local environment in which the symbol `loop` is visible (though initially unbound). `loop` is then assigned (by the `set!`) the value of the `lambda` expression that follows. Inside the body of this `lambda` expression the symbol `loop` occurs—why isn't this circular reasoning? Simply because the **value** of `loop` isn't required until the `lambda` procedure is run, which first occurs in the invocation (`loop 1`) that follows the `set!`. After the `set!`, `loop` is bound to the `lambda` procedure itself.

7. MULTIPLE VALUES

XLisp borrows from Common Lisp the ability of a procedure to return multiple values. This is potentially more efficient than returning a list of values—the list must be CONSed together and then destructured at the receiving end. One can create multiple values using the procedure **values**, that takes any number of arguments and returns them all (as values) in the same order:

```
[1] (values 1 2 3)
1
2
3
[2] _
```

One can also use the functions **values-list** that takes a list as its single argument and then returns all the list's elements as the multiple values:

```
[2] (values-list '(a b c))
a
b
c
[3] _
```

Of course to make any use of this one needs forms that capture multiple values. The simplest is **multiple-value-set!** whose first argument is a list of symbols and whose second argument is any expression returning the appropriate number of multiple values:

```
[3] (multiple-value-set! (p q r) (values 10 20 30))
30
[4] p
10
[5] q
20
[6] r
30
[7] _
```

More useful is the special form **multiple-value-bind**, whose syntax in the XLisp Manual is given as:

```
(multiple-value-bind (var ...) vexpr expr ...)
```

Here **(var ...)** is the list of symbols to be bound, exactly as in the first argument **multiple-value-set!**. The difference is that **multiple-value-set!**, like **set!**, looks for its symbols starting with the innermost enclosing lexical environment and modifies the first instance of the symbol it finds—or, finding none, adds the symbol to the global environment. **multiple-value-bind**, on the other hand, behaves like **let**: it creates a new nested environment and introduces the symbols there. It is in this nested environment that the arguments following **vexpr** are evaluated, sequentially as with **let**, with the value of the final **expr** being the value of the form.

For example:

```
[7] (multiple-value-bind (p q r) (values 1 2 3)
      (format #t "p = ~A~%" p)
      (format #t "q = ~A~%" q)
      (format #t "r = ~A~%" r))
p = 1
q = 2
r = 3
()
[8] (list p q r)
(10 20 30)
[9] _
```

This shows that `p`, `q`, `r` have been defined locally within the scope of the `multiple-value-bind`, and that their global values are unaffected (as seen in [8]).

It is possible to return **no values** whatever, by simply invoking `values` without any arguments:

```
[9] (values)
[10] _
```

This allows us to eliminate the distracting printing of return values from calls that are invoked for printed output—as for example in the final `()` that follows the three printed lines produced by [7]. This `()` results from the final `format` call, which returns `()` as its value, which then becomes the value of the `multiple-value-bind` call and which is dutifully printed by the READ-EVAL-PRINT loop in the standard way. Observe:

```
[10] (display "Printed Output")
Printed Output
#t
[11] (begin (display "No distractions!") (values))
No distractions!
[12] _
```

8. REFERENCES

[1] **XLISP: An Object-Oriented Lisp**, Version 3.0, David Michael Betz

This is the XLisp Manual referred to in these notes; it is the definitive reference for XLisp.

[2] **The Little Schemer**, Daniel P. Friedman and Matthias Felleisen

Formerly distributed as The Little LISPer, this volume is a simplified introduction that employs a question/answer format that facilitates self-pacing. The content is solid, and good insights are developed through well-chosen examples.

[3] **The Seasoned Schemer**, Daniel P. Friedman and Matthias Felleisen

A sequel to The Little Schemer.

[4] **Structure and Interpretation of Computer Programs**, Harold Abelson and Gerald Jay Sussman, with Julie Sussman

A classic computer science text, now in its second edition, that uses Scheme as a vehicle for illustrating powerful and fundamental ideas.

[5] **Scheme and the Art of Programming**, George Springer and Daniel P. Friedman

An excellent text for learning, not only the mechanics of Scheme programming, but the idiom of thought as well. The well-developed illustrations demonstrate the expressive power of the language.

[6] **The Scheme Programming Language**, Second Edition, R. Kent Dybvig

A reference book that covers ANSI Scheme. XLisp is not identical to Scheme; none-the-less, this volume can be helpful.

[7] **XLisp Tutorial: an Introduction for C Programmers**, Matthew Halfant

This manual. All lisps exploit recursion, so the self-reference is thematic.

9. INDEX

Words in **bold** are XLisp commands

&

&key ... 21
&optional ... 20
&rest ... 20

,

'count ... 16

A

append ... 18

C

car ... 16
cdr ... 17
class ... 26
cond ... 15
cons ... 17
count ... 14

D

data type
  Boolean ... 6
  character ... 6
  class ... 6
  integer ... 5
  number ... 5
  object ... 6
  procedure ... 6
  string ... 6
  symbol ... 6
  vector ... 6
define ... 8
display ... 22

E

environment
  global ... 9, 14
  local ... 14
  nested ... 9
  outer ... 9

F

first ... *See* **car**
Flow of control functions
  if 8
format ... 22, 31
fourth ... 16
function ... 15

I

if 8
Input/Output functions
  display ... 22
  format ... 22, 31
  read ... 13
  write ... 22

K

keyword
  lambda ... 14

L

lambda ... 14
length ... 17
let 9
let* ... 11
list ... 6, 16
List functions
  append ... 18
  car ... 16
  cdr ... 17
  cons ... 17
  destructive ... 18
    append! ... 18
  first ... 16
  fourth ... 16
  length ... 17
  list ... 16
  list-ref ... 16
  rest ... 17
  reverse ... 17
  second ... 16
  third ... 16
list-ref ... 16

M

multiple-value-bind ... 30
multiple-value-set! ... 30

P

print ... 10
proc ... 23
procedure ... 14

Q

quote ... 12

R

read 13, 22
reset 12
rest ..... *See* **cdr**
reverse 17

S

second 16
S-Expression 6, 28
sqrt 15
str 23
symbol table 9
  global 9

T

third 16

V

values 30
values-list 30

W

write 22

Z

zero? 9