CS320 LISP Handout

On most servers and workstations we have a small LISP interpreter called XLISP. On most lab machines we have Gnu's version of Scheme -- *guile*. This document is a introduction to XLISP. It has just enough LISP for CS320 laboratory sessions. Nearly all of the original LISP functions/programs work with *guile* but the details of editing and loading functions/programs are a little different -- see the Guile handout. XLISP "programs" should be put in files with names that end ".lsp". Guile programs should be in files with names ending in ".scm" or ".scheme".

Starting XLISP

To start XLISP type in the UNIX command

```
xlisp
```

or

```
xlisp filename...
```

The command 'lisp' may run a more complex Allegro LISP not XLISP.

Stopping XLISP

The interpreter quits when the user inputs an end-of-text character -- CTRL/D on UNIX -- or when LISP tries to evaluate the expression

```
(exit)
```

Notice that the parentheses in (exit) are needed. Otherwise LISP sees a variable with no value!

What Happens

When you run XLISP:

XLISP reads and executes some library files, and then any files you have listed.
LISP takes over the shell window.
You start interacting with LISP.

From then on until you stop XLISP (exit) this what happens:

You input LISP *expressions*.
The *interpreter* reads *expressions* and *evaluates* them.
It responds to every *expression* by producing a *value.*

So in XBNF:

LISP.input::=#*expression*.
LISP.output::=#*value*.

Each *expression* corresponds to a single *value*.

Expressions

Almost all *expressions* apply an operator to some arguments. This is written like this:

(*operator arguments*)

A simple example is

```
(-    1)
```

which outputs -1. All LISP expression except *variables* and some *constants* (numbers and strings) use parentheses. More complex expressions also have the operator/operation first. For example
the following adds 2 and 3 and outputs 5:

```
(+ 2 3 )
```

The arguments can be any *expression*. The following stands for *b***b*-4**a***c*:

```
(- (* B B) (* 4 A C) )
```

The parentheses tell the interpreter where expressions begin and end.

Values

The idea behind LISP is to make LISt Processing easy. All LISP data is expressed as a list. A *list* is something like these:

```
(THE CAT SAT ON THE MAT)
(TOM (SPECIES CAT) (LEGS 4) (FUR BLACK) (GENDER MALE) (SIZE LARGE))
(+ (* X X) (* Y Y) (* Z Z))
```

Notice that LISP expressions are also *lists.*

Programs

All LISP code - programs, commands, subprograms, etc., are written as functions. These functions are encoded as *lists*. LISP expressions define new functions that the user can use to solve problems. In LISP, functions are programs and all programs are functions. See the syntax later.

An XLISP program file defines a series of functions. Each is a program the user can run. The file should have a name that ends '.lsp'. When the XLISP program runs, it can load '.lsp' files. It then waits for the user to input an expression to evaluate. This will normally be an expression calling one of the newly loaded functions. Here is a simple (useless) example of DEfining a FUNction:

```
(DEFUN hello() "Hello, World!")
```

This lets the user input `(hello)` and get "`Hello, World`!" back.
Here are three functions (a, b, and c) (with comments) concerned with the Hailstone series of integers:

```
(DEFUN a (n) (1+   (* 3 n))); multiply by 3 and add 1
(DEFUN b (n) (/ n 2)); divide by 2
(DEFUN c (n) (IF (oddp n) (b (a n)) (b n)) )); if odd do a and b else do b
```

This allows the user to "play" with the sequence of numbers by inputting commands(expressions) like this

```
(C (C (C 17)))
```

Editing and XLISP
It is not possible to edit a function inside XLISP (but see programming projects at the end of this handout). A simple way to develop XLISP programs is to use 'vi' or Emacs in one window and run XLISP in a different window. Don't forget to save before you run XLISP!

In '*vi*' the '%' command jumps between matched parentheses. Use it to check your typing. Also in *vi* the command

```
:set showmatch
```

makes '*vi*' show you how many parentheses you have closed....try this, you may like it.

Syntax

LISP uses a simple, rigid, and unambiguous syntax called "Cambridge Polish" after the town of Cambridge in Massachusetts:

expression::=*constant* | *variable* | "(" *operator* *#expression* ")".

Lexemes

word::= # *word_char*,
word_char ::=`any character except spaces, parentheses, quotes, and commas`.
Note: a word can include a period.

number::= # *digit* O("." # *digit*), -- most modern LISPs allow floating point numbers as well as integers.
digit ::= "0" .. "9".
string::= *quote* #(*char* ~ *quote*) *quote*.

LISP is not case sensitive outside strings. White space is used to separate adjacent words and otherwise is significant only inside strings. For example (*12 ....) has an operator "*12" and (* 12 ...) has the operator "*". The meaning of a *word* is determined by its place in expressions - it may be part of a constant, a function name, or a variable.

Constants
Constants have the property that evaluation leaves them unchanged:

constant::=*number* | *string* | "NIL" | "T" | *quoted_list.*
NIL is used for False and is short for the empty list (),
T is a predefined atom used for True,
quoted_list::= "(QUOTE" *list*")" | "'" *list*.

```
(QUOTE THIS IS A CONSTANT LIST)
```

The QUOTE operator stops the interpreter from evaluating QUOTE's arguments. So (+ 1 2) has value 3 but (QUOTE (+ 1 2)) has value (+ 1 2). The single `blip` "'" is used as an abbreviation: '(+ 1 2) ia short for (QUOTE (+ 1 2)).

Lists

A list can be empty (NIL), an atom (word, number, string), a pair, or a sequence of lists between parentheses.

list::= NIL | *atom* | CONS_*pair* | *linked_list*.

atom::= *string* | *number* | *word*.

123

CAT

"CAT"

CONS_pair::= "(" *list* "." *list* ")", a constructed pair.

(A . B)

linked_list::= "(" #*list* ")".

(A B C)

The meaning of a list depends on where it appears in a program.

Variables

LISP allows you to bind a value to an *atom*. You can then use the atom in place of the value. The effect is that the atom has become a *variable*. The binding is done when a function is called. The formal parameters (arguments) of a function become local variables that are bound to the *values* of the actual parameters. There are some (impure) operators that change the value bound to an atom.

variable::=*word*.

Notice that inside a QUOTE an atom stands for itself and is never treated as a variable. Also notice that an atom is not treated as a variable if it follows a parenthesis. After an "(" the first *word* is the name of an operator/function.

Operators

Expressions often start with an operator like this:

(*operator argument*1 *argument*2 *argument3* ...)

Operators are classified as *functions* (normal) and *macros* (peculiar):

operator::= *function_name* | *macro_name*.

argument ::= *expression*.

Functions and macros have the same kinds of names. The name can be any word: ADD and + for example. The meaning bound to the word determines whether it can be a function, a macro, or a variable.

Functions evaluate all their arguments before they start to work. Addition (+) and multiplication (*) are typical functions. Functions created with DEFUN and DEFINE are like this too. Other operators are called *macros* or *functional forms*. QUOTE, DEFUN, COND and IF are examples of macros. Macros use pass by name and must explicitly evaluate their arguments in their definitions.

Note

The meaning of a word often depends on its context (where it is placed):

Inside a QUOTE	a constant
After a parenthesis	an operator
By itself	a variable

For example (QUOTE +) outputs +, (+ ....) adds up the arguments, and + gives an error.

Predefined Functions and Forms

XLISP has *many* ready made functions. We keep a complete list online. There is a partial list in this handout. The most important functions are in capitals. Never waste time defining a function if it already exists.

Some predefined LISP Functions

Notation: Optional arguments are in square brackets. *e* is an expression, *t* any condition, *f* any functions, *s* is a symbol, *n* is an arithmetical expressions. Some functions/forms are **CAPITALIZED BOLD** and must be mastered.

Arithmetic expressions

(***** *n*...) multiply (+ *n*...) add
(**-** *n*...) subtract, negate (/ *n*...) divide
(1+ *n*) add one (1- *n*) subtract one
(abs *n*) the absolute value of a number
(cos *n*) (sin *n*) (tan *n*) trig functions
(exp *n*) exponential of *n*
(expt *n*1 *n*2) *n*1 to the *n*2 power
(float *n*) converts to a floating point
(max *n*...) (min *n*...) the largest/smallest
(random *n*) random number between 1 and n-1
(rem *n*...) remainder after division
(sqrt *n*) compute the square root of *n*
(truncate *n*) truncates float *n* to an integer

Relational expressions

(/= *n*1 *n*2) test for not equal to
(< *n*1 *n*2) test for less than
(<= *n*1 *n*2) test for less than or equal to
(= *n*1 *n*2) test for equal to
(> *n*1 *n*2) test for greater than
(>= *n*1 *n*2) test for greater than or equal to

Boolean Expressions/Predicates

NIL Empty list used to indicate **false** in tests
T predefined constant used for **true**.
Any non-**NIL** value is treated as **true** in tests.
(**ATOM** e) is this an atom?
(consp e) is this a non-empty list?
(eq e1 e2) (eql e1 e2) (equal e1 e2)
Various equality tests.
(evenp *n*) is *n* even?
(listp e) is this a list (not a null or atom)?
(member e1 e2) is e1 an item in e2?
(minusp *n*) is *n* negative?
(**NULL** e) is this an empty list?(sometimes NULLP)
(numberp e) is e a number?
(oddp *n*) is *n* odd?
(plusp *n*) is *n* positive?
(**ZEROP** *n*) is *n* zero?
(**AND** [e]...) a logical and (short-circuited)
(**NOT** e) is e false? = (null e)!
(**OR** [e]...) logical or (short-circuited??)

List Expressions

(**APPEND** [e]...) append lists
(assoc e1 e2) Find pair (*x y*) in e2 with *x*=e1,and return y.
(**CAR** e) return the first item of CONS-pair e
(**CDR** e) return the second item of Cons-pair e
(**CONS** e1 e2) construct a new Cons-pair
(*cxxr* e) cadr, caar, cdar, cddr
(*cxxxr* e) all *cxxxr* combinations(x::=a|d)
(*cxxxxr* e) all *cxxxxr* combinations
(last e) return the last node of list e
(length e) the length of a list or string
(**LIST** [e]...) constructs a list of values
(**NTH** *n* e) the *n*th item of e
(nthcdr n e) the *n*th cdr of e
(reverse e) reverse a list

Strings and Characters

(char e1 n) *n*th character in string e1
(strcat [e]...) concatenate strings

Functional

(apply f args) apply a f to args
(funcall f [args]...) call function with args
(mapc f e) (mapcar f e) (mapl f e) (maplist f e)
apply f to all (cars, cdrs, elements) in list e

Printing

(prin1 e ...) print e
(princ e) print without quoting
(print e) print e on a new line
(terpri) terminate the current print line
(write-char character) write a character

Some Macros

Quotation

'e::=(**QUOTE** e) return e unevaluated
#'e::= (function e) quote a function
#(e...) ::= an array

Lambda Expressions

(**LAMBDA** (args) *e*) a function from args to *e*.

Assignment

(setq s e) set the value of a symbol s to e

Selection

(**COND** [pair]...) select first pair=(t e) with *t* non-null and evaluate *e*.
(case *e* [*case*]...) select by case
case::=(*value* [e]...)
(if *e1 e2* [e3]) if *e1* then *e2* else *e3*

Iteration

(do ([binding]...) (t) e...) loop
(do* ([binding]...) (t) e...) loop
(dolist (s list) e...) for s in list
(dotimes (s e_1) e_2...) s in 0..e1-1

Blocks

(**LET** ((s e1)...) e2...) block{LIST s=e1;... .e2..}
(progn [e]... e_n) execute sequentially{...}
(prog ([binding]...) [e]...) begin...end/{...}

Control

(**EXIT**) exit XLISP
(return e) cause a *prog* to return value

Files

(LOAD filename [vflag [pflag]]) load a file

Control Structures

LISP has the usual complement of control structures: loops, selections, compound statements but *they are all expressions and return a value*. Here is the most important two

Selection

(COND (*e1 e2*) *more_pairs*) =

If the value of *e1* is	Not NIL	NIL
then return value of	*e2*	(COND *more_pairs*)

(IF *e1 e2 e3*) ; if *e1* then *e2* else *e3*
=(COND (*e1 e2*) (T *e3*))

Iteration

LISP style uses recursion rather than iteration but XLISP does have loops called:
DO DO* DOLIST DOTIMES

Defining New Functions

Functions are declared in two ways in LISP:
(DEFINE (*name args*) *expression*)
(DEFUN *name* (*args*) *expression*...)
Both create a global meaning for the name and allow it to be used as an operator:
(*Name actual_arguments*)

In the CSUSB version of XLISP we have both DEFINE and DEFUN-- but they are subtly different. See later.

Axioms

LISP is "logical" in the sense that its inventors wrote down a set of assumptions (axioms) that can be made about LISP expressions. They tried to make LISP implement the axioms. As a result you can predict the result of evaluating an expression mathematically. Here are some of the axioms:

For all LISP expressions x,y:
(CAR (CONS x y)) = x
and (CDR (CONS x y)) = y.

If *x* is a Cons-pair then
(CONS (CAR x) (CDR x)) = x.

If the value of c is not NIL then (COND (c x) y) = x.
If the value of c is NIL then (COND (c x) y) = (COND y).

Abbreviations:
(CADR x) = (CAR (CDR x)),
(CDDR x) = (CDR (CDR x)),
(CDAR x) = (CDR (CAR x)),
(CAAR x) = (CAR (CAR x)),
and so on for CADDR, CADDDR, CDDDR, ...

(NTH 0 x) = (CAR x)
For n>0,
(NTH n x) = (NTH (- n 1) (CDR X))

Hints

(1) **Let LISP do the work for you.**

(1a)Forget efficiency! Quickly break down complex problems into simple ones (top-down) in the *clearest* way you can. Then code and test solutions to the simple problems. Uses the solutions to parts of the problem to solve larger problems. Once they work stop. If the user complains you can *always* speed up any program.

(1b)LISP **is** the main program! The user is then asked which function to call. Aim for lots of useful but small functions. Give the user a wide choice of "programs" to run. Use them yourself in your code.

(1c)LISP functions are often recursive.

(1d)Don't waste time on input/output. LISP functions are given lists as data (by the interpreter) and output a list to the standard output.

(2)**Forget assignments, sequences, and loops!** Look for ways to classify the givens (COND, ATOM, NULL, MINUSP...) plus ways to decompose them (CAR, CDR,...) plus ways to assemble what is needed (+, CONS, LIST, APPEND, ...)

(3) **Divide and conquer.** Look for known functions: predefined, defined later, or the function being defined.

(4) **Layout LISP** and **indent** sub-lists. Use comments. Comments start with a semicolon ";" and go to end of line.

```
(DEFINE (length L); Number of items in list L
   (COND
     ((NULL L)  0) ; an empty list has zero items
     ((ATOM L)  1) ; an atom has one item
     (   T     (+ 1 (length (CDR L)) )) ; L has one more item than its tail
   );COND
);DEFINE
```

(6) Do you want to store some results in a variable? Introduce a function with arguments to store them! For example suppose I need a function to solve a quadratic equation:

(SOLVE 1.0 2.0 1.0)

(SOLVE *a b c*); print solutions to $a\,x^2+b\,x+c=0$

Now the value of the *discriminant* ($b^2-4\,a\,c$) and the *denominator* ($2\,a$) are used several times in the formulae, so we might define three functions as follows:

```
(DEFINE (SOLVE A B C) (SOLVE2 A B C (DISC A B C)(/ 1.0 (* 2.0 A)) ) )

(DEFINE (DISC A B C) (- (* B B) (* 4.0 A C)))

(DEFINE (SOLVE2 A B C D DEN)
  (COND
    ( (> D 0) (LIST
                (* ( - (- B) (SQRT D)) DEN)
                (* ( + (- B) (SQRT D)) DEN)
                );LIST
      ); >
      ( (= D 0) (* (- B) DEN)
      ); =
      ( T         (CONS
                    (* (- B) DEN )
                    (* (SQRT (- D)) DEN )
                  );CONS
    );T
 );COND
);DEFINE
```

(7) When the interpreter doesn't do anything after typing in an expression or definition..... add right parentheses.

(8) End all CONDs with an '**or-else**' pair like this (T something).

Semantic Details

Functions and Scopes
Each function call adds new local variables to the environment by (1) evaluating the actual arguments, and (2) binding them as values to the formal arguments. These local assignments are removed when the function returns.

A **LAMBDA** expression like

(**LAMBDA** (*a1 a2 a3 ... an*) *e*)

represents an unnamed function that expects to be given *n* arguments. These will be expressions that are evaluated and bound to the formal arguments as values. This is a complete new set of variables a1..an. The formal arguments a1..can appear in the expression e. Expression e is evaluated to give the value of the function. The variables a1...an are then removed and the value is returned. Notice that any older versions of the variables now re-appear.

The first LISP was defined by an interpreter - called EVALQUOTE. It used dynamic scoping with shallow binding and associated a "P-list"(Property List) with each atom. Nowadays LISPs (including Scheme) use static scoping. P-lists are still a part of LISP thinking and programming, however.

XLISP provides both old fashioned functions and modern statically scoped ones.

I added the

(DEFINE (*name args*) *expression*)

macro to XLISP. I stayed close to the original EVALQUOTE model and used the name's P-list to bind the name to a LAMBDA expression. The result is that 'define' uses *dynamic* scoping. After

(DEFINE (*N args*) *e*)

N means (LAMBDA (*args*) *e*). It is called like this (*N args*). But it is shorthand for the function body(LAMBDA). So it has dynamically scoped variables. Because *e* is evaluated when called it's non-local variables are interpreted in the environment of the call... not the definition.

In our LISP (XLISP) the

(**DEFUN** *name* (*args*) *expression*)

macro defines the function and ensures static scoping by binding the environment of the definition into the function code. This creates something called a *closure* that combines details of the definition and its environment. After

(**DEFUN** *N* (*args*) *e*)

N is bound to a kind of a *compiled* function called a closure. It is called like this (*N args*). The compilation occurs in the environment of the definition and so non-local variables refer those *defined at the time and place of definition.* In short DEFUN uses static scoping.

This becomes important for encapsulating data by a collection of functions.

The LISP has a block structure created by the **LET** function/form:

(**LET** ((*variable value*)...) *expression*...)

introduces a set of local *variable*s each with an initial *value*. Each expression is executed in turn and can use the local variables. For example:

```
(LET ( ( X 1) (Y 2) ) ( + X Y))
```

returns 3, the sum of 1 and 2. The variables can not be accessed outside the (LET...) block.

If an expression in the LET is a (DEFUN ...) then it defines a function that *has access to the local variables*. Several DEFUN's give several functions that share the variables. The function name has *global scope* and so the function can be called from outside the (LET ...). So, the LET has created an object with attributes (the local variables) and member functions/operations to access them. For example here is a simple counter that starts at 0 and is increased by 1 each time (up) is called. (val) returns its value:

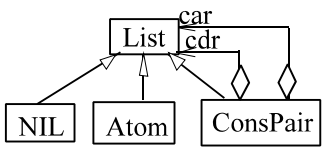
```
(LET ((c 0)) (DEFUN up () (SETQ c  (1+   c)))  (DEFUN val () c) )
```

Implementing LISP

The data in LISP is a list. Internally all lists are either empty, atomic or a CONS-pair (Constructed Pair)

list ::= *nil* | *atomic* | *Cons-pair*.

Internally these are (1) the null pointer, (2) a pointer to a string or a number, (3) an object with two pointers called '**car**' and '**cdr**' respectively.

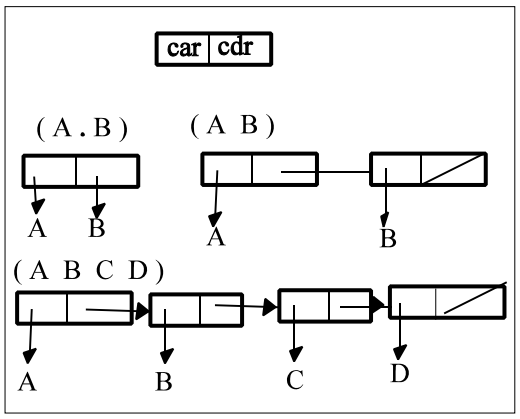


A CONS-pair is constructed by the CONS function:

(CONS 'A 'B)

This is a pair of pointers (the *car* and the *cdr*) that point to atoms A and B respectively, See the diagram.
The value is printed with a **dot** (**.**) like this:

(A . B)



It is therefore called a "Dotted pair".

All lists (for example (A B C D)) are constructed from dotted pairs. For example the expression

(LIST 'A 'B 'C 'D)

constructs this data structure

(A . (B . (C. (D . NIL))))

which might be stored in RAM as shown in the table.--->

Name/Address	Contents		
1234	123	1242	
1242	126	1255	
1255	128	1273	
1273	1231	**0**	
123	A		
126	B		
128	C		
1231	D		

The address (1234) is returned as its value. When printed you see the following however:

(A B C D)

See Also

There are many examples/tutorials/experiments that can be found from the CS320 resources page:

http://www.csci.csusb.edu/cs320/resources.html#LISP

Here is a page of pointers to many WWW pages on LISP:

http://www.csci.csusb.edu/dick/samples/lisp.html

The source code and documentation for CSUSB XLISP is online:

http://www.csci.csusb.edu/cs320/lisp/src/

It is easy to port to other operating systems. You may download, compile, use, and change this code for any non-profit purpose. Do not sell it - improve it and give it away.

Review Questions

1. What does a user input to a LISP interpreter? What does the interpreter do with the input? What does it output?

2. How do you terminate XLISP? What are the parentheses in this input for?

3. What is the first character and the last character of all expressions that have an operator? What follows the first character? What comes next?

4. Is LISP case sensitive? Is white space significant and if so, where? Name some characters that can be in a string but not in an word.

5. What is the value of LISP expressions: (+ 1 2), (* 3 (+ 1 2)), and (* (+ 1 2) (+ 2 5)).

6. Write the C/C++ expressions in LISP: 1+2, 1+2*3, sqrt(4).

7. Consider a word like ALPHA. When it appears as the input expression to a LISP interpreter what does the interpreter expect it to be? How is it interpreted here: (ALPHA .....)? How about here: (.....ALPHA ....)

8. Fill in the blank in: Just about everything in LISP is a _____ structure.

9. What are the values of (QUOTE (+ 1 2)), '(+ 1 2), (+ 1 2).

10. What happens if you don't close all the parentheses in a LISP expression as you input it?

11. What is the difference between a function and a macro in XLISP? Give an example of each.

12. Name two operators that are used to create new functions.

13. List half-a-dozen symbols that are used to represent arithmetic operators in XLISP.

13. List the 6 relational operators of XLISP.

14. What do the following operators do in XLISP: ATOM, NULL, ZEROP?

15. List the three boolean operators and the two Boolean values that are predefined in XLISP.

16. What is the CAR and the CDR of (x . y)? What is the CAR and the CDR of (*x y*)?

17. What are the values of (CAR (CONS x y)) and (CDR (CONS x y)) respectively?

18. Explain the difference between the dotted pair (A . B) and the list (A B).

19. How are the lists (A), (A B), and (A B C) expressed using dotted pairs (___ . _____) and NIL?

20. What is COND short for? Suppose *x*, *y*, *z* are LISP expressions, explain -- step-by-step -- how the following expression is evaluated in LISP: (COND (*x y*) (T *z*)).

21. Express (CADR x) and (CADDR x) in terms of CAR, CDR and expression x.

22. Consider (DEFINE (DELTA A B C) (- (* B B) (* 4.0 A C))). What is the name of the function? How many arguments does it have? What is (DELTA 1 2 3)? Rewrite the DEFINE using DEFUN.

23. Consider (DEFUN DIFF (X Y) (ABS (- X Y))). What is the name of the function? How many arguments does it have? What is the value of (DIFF 1 2)? Rewrite this DEFUN using DEFINE.

24. What parameter passing methods are used in LISP functions?

25. Study this: (LET ((X 3)(Y 2)) (* X Y)). What variables are declared and what are their values? What value is printed if the above is input?

26. What do you use the operator SETQ for?

27. Write the expressions (-b + sqrt(b*b - 4 a c))/(2*a) and (-b + sqrt(b*b - 4 a c))/(2*a) as LISP expressions.

28. Write the following C/C++ expression as a LISP COND expression: (delta(a,b,c) > 0? two_roots(a,b,c): two_parts(a,b,c))

Programming Projects (optional)
1. Add a macro (EDIT function_name) to our XLISP by hacking the source code

2. Find out about these two famous LISP programs: Doctor and Eliza.