Common LISP-Introduction

1. The primary data structure in LISP is called the s-expression (symbolic expression). There are two basic types of s-expressions: atoms and lists.

2. The LISP language is normally implemented with an interpreter.

3. Evaluation of an s-expression involves performing a computation on it. The result of evaluating an s-expression is called the value of the expression.

4. Simple arithmetic:
   a. \((+\ 3\ 4)\)
   b. \((\times\ 3\ 4)\)
   c. \((\times\ (+\ 8\ 3)\ (+\ 4\ (*\ 9\ 2)))\)
   d. \((-\ 8\ 3)\)
   e. \((/\ 8\ 2)\)
   f. \((/\ 8\ 3)\);; This will give 8/3 as an answer. One value must be floating pt. in order to get a real-valued result. (float (/ 8 3)) also works.
   g. \((\times\ 2\ 3\ 4)\)
   h. \((1+\ 4)\)
   i. \((1-\ 5)\)

5. Lists are sequences of s-expressions inside matching parentheses

6. Atoms are represented by numbers or symbols (sometimes called literal atoms). Symbols are used to represent variables and function names.

7. LISP tries to evaluate every s-expression it sees! After evaluation, the interpreter prints the value that has been computed.

8. Rules of evaluation
   a. Numbers evaluate to themselves
   b. Symbols evaluate to the last value assigned to them
   c. Lists are evaluated by interpreting the first element as a function name and the rest of the list as arguments to that function. The arguments are usually evaluated and the function is then applied to the resulting values. The value returned by the function is the value of the list.
9. Using the SETQ function

a. The **setq** function as well as the **set** function is used to perform simple assignment of a value to a symbol. The assignment is global and using setq and/or set is usually frowned upon by most avid LISP programmers. Setq does not evaluate its first argument whereas set does. Some examples follow.

b. Examples

   (setq x 5)
   (Setq x ‘5)
   (setq y (quote (a b c d _))) ; quote is a function of one argument which is not evaluated
   (setq y ‘(a b c ) ) ; This is a shorthand notation for the quote function
   (setq value ‘artificial-intelligence)
   (set ‘value2 ‘computer-science)
   (set ‘value3 data structures) ; This assignment would result in an error since data-structures has not been assigned a value at this point

10. Handling errors

   If XLISP gives you an error message, type CTRL-C to return to the top level of the LISP environment.

11. car and cdr

   a. Examples of car and cdr functions. First and rest can also be used.

      ( car ‘(a b c ))
      ( cdr ‘(a b c ))
      ( setq x ‘(a b c))
      (car x )
      (cdr x )
      x
      ( car ( cdr ‘(a b)))
      ( cdr ‘(( a b ) ( c d )))
      ( cdr ( car ( cdr ‘(( a b c) ( d e f )))))

   b. Using cadr etc.

      ( car ( cdr ‘(( a b c ) ( d e f ))))
      ( cadr ‘((a b c ) (d e f ))) ; same as directly above
      ( cadadr ‘( a b c ) (d e f )))
      ( car ‘( cdr ( a b c ))) ; Here cdr is just a constant
c. The empty list
   (cdr '(c)) ;; This evaluation will return NIL
   () ;; This is also NIL
   () ;; as is this
   nil ;; this is also NIL

**cons**

Cons is a function of two arguments. The second argument should always be a list. Cons returns as its value the list obtained by taking the second argument and sticking the first one in front of it.

d. (cons 'a '(b c)) ; The result here is (a b c)
e. (cons 'a (cons 'b '(c d))) ; result is (a b c d)
f. (cons 'a nil) ; result is (a)
g. (setq x 'a)
h. (setq y '(b c))
i. (cons x y)

cons constructs a list and therefore the result requires extra storage. Cons is an expensive operation

12. list and append. List construction functions

   a. (list 'a 'b 'c) ; (a b c)
   b. (list 'a '(b c) 'd) ; (a (b c) d)
   c. (append '(a b) '(c d) '(e f)) ; (a b c d e f)
   d. (append '(a b) nil) ; (a b)

13. A form is a symbolic expression intended for evaluation. A constant is a form whose value is predetermined. Some special symbols like nil are constants, as are all numbers and quoted expressions.
Defining functions

1. (defun addthree (x)
   ( + x 3))
   The above defines a simple function that will add three to the integer value passed to it. DEFUN is the function defining function in LISP. It is a special function that does not evaluate any of its arguments. It merely associated the formal parameter list and bodies of code with the function name for future reference. The name of the function above is ADDTHREE. After the function name comes the formal parameter list which in this case just contains x. finally, we have the body of the function which in this case is simply a list representing a call to the + function. LISP functions always return a value. DEFUN is no exception. The value returned by DEFUN is the name of the function, in this case ADDTHREE. However DEFUN is not used here for the value it returns, rather it is used for its side-effect of creating a new function.

2. Define a function xcons that reverses the argument order in a call to the function cons.
   ( defun xcons (1 e)
     ( cons e 1))
   Call the function:
   ( xcons ‘(b c) ‘a) ; (A B C)

3. Define a function whose input is a list of two elements and whose output is a list of the list of each element

   Example >>> (list-of-lists ‘(a b))
   ( (a) (b))
   Definition >> (defun list-of-lists (x)
     ( list ( list ( car x)) ( list (cadr x))))

4. Formal parameters

   (setq x ‘(a b c)) ; the value of x is set to a list. This assignment is on the top level of LISP

   ( defun test (x)
     ( test2 ( + x 3))) ; two functions with the formal parameter x are defined

   ( defun test2 (x)
     (* x x)) ; now call the function test –

   ( test 5) ; this call will return the value 64 ; however the value of the parameter x will ; still be (a b c)
LISP does not allow parameters from one function call to interfere with those of another, you may use the same symbol as a parameter in as many definitions as you like. One way to think about this is that each time a function is called, LISP creates a new variable for each of that function’s formal parameters. Within that function call, all references to a given formal parameter are interpreted as references to the variable created just for that function call.

5. Free variables, global variables

A variable designed by a symbol at the top level of LISP is called a GLOBAL VARIABLE. (usually denoted by a *name*) A variable used within a function, but which is not a formal parameter of that function is called a free variable within that function.

An example

(setq sum '(a b c))

(defun sum-average (x y)
  (setq sum (+ x y))
  (/ sum 2))

(defun sum-average-caller (sum x y)
  (sum-average x y) sum)

; now call the function sum-average-caller

(defun sum-average-caller 0 55 65)

>>> the returned value will be 0
\(\rightarrow\) sum
the value of the global variable sum will be 120!!!

The problem of which variable is referenced by the occurrence of a symbol in a program is referred to the Scoping Problem. Essentially there are two types of Scoping:

Lexical Scoping and Dynamic Scoping

Common LISP supports lexical scoping and most implementations of common LISP are lexically scoped. Having lexical scoping means that formal parameters can be referenced only by the code of the functions within which they are parameters.

With dynamic scoping, variables designated by a parameter can be referred to outside their functions, and the most recently created variable of a given name is used as the reference for a symbol. In the above example, the call to (sum-average-caller 0 55 65) would return 120 but the value of the global variable sum would remain (a b c).
Predicates, Conditionals, and Logical Operators

1. Predicates are functions that return true or false. In LISP false is indicated by the atom NIL. NIL also represents a valid list expression. Here are some useful predicates

   a. ATOM returns true if the argument it is passed is an atom.
      ( atom 'a) ; true
      ( atom '(a b c)) ; Nil

   b. LISP returns true if its argument is a list
      ( listp '(a b c))
      ( listp (car '(a b c))) ; Nil

   c. NULL returns true if its argument is nil. It returns nil otherwise
      ( null nil) ; true
      ( null 'a) ; false
      ( null () ) ; true

   d. EQUAL returns true if two s-expressions look alike
      ( equal 'a 'a)
      ( equal '(a b c) '(a b c))

   e. NUMBERP returns true if its argument is a number

   f. Other predicates that take numbers as arguments include
      ZEROP ODDP EVENP

   g. MEMBER takes two arguments, the second of which must be a list. MEMBER returns the part of the list of which the mach occurs if its first argument is a member of the list specified as the second argument
      ( member 'b '(a b c)) ; returns (B C) not true!!
      ( member 'b '(a (b ) c)) ; returns nil
      ( member '(a b) '(a d (a b))) ; returns ( (a b) )

   h. There are many more LISP predicates but the above list is representative of the most commonly used predicates.
2. Conditionals

a. The COND function is traditionally used to perform conditional operations

b. COND’s general form

```
( cond
  ( expr11  expr12  expr13 ...)
  ( expr21  expr22  expr23 ...)
  ( expr31  expr32  expr33 ...)
  ...
  ...
  ...
  ( exprn1  exprn2  exprn3 ...)) ; end of cond clause
```

LISP returns the value of the last expression it evaluates as the value of the COND

; let’s define a function using cond

```
(defun test-val (arg)
  (cond
    ((null arg) 'empty-argument)
    ((listp arg) (cdr arg))
    (t '(arg is an atom))) ; end of cond
  ) ; end of test-val
```

3. Logical Operators

The logical operators include AND, NOT and OR

Examples:

```
(not (atom x)) ; returns true if x is not an atom
(and (evenp x) (< x 100) (evenp y)) ; returns true if all 3 conditions are true
(or (null x) (numberp x)) ; returns true if x is null or if x is a number
```
ASSIGNMENT FORMAT

Create a file that will define and test all of the functions as follows:

;; Your Name
;; Problem 1

(defun length (l)
;; This function finds the length of a list of atoms
;; If the list is empty, the function returns zero.
(cond
  ((null l) 0)
  (t (+ 1 (length (cdr l))))))

;; Problem 2

(defun drop (n l)
;; The function creates a new list that has the
;; first occurrence of n removed from the list
;; (drop 'a '(x y a b a c)) returns (x y b a c)
(cond
  ((null l) nil)
  ((equal n (car l)) (cdr l))
  (t (cons (car l) (drop n (cdr l)))))))

;; Problem 3

(defun drop-all (n l)
;; The function creates a new list that has the
;; all occurrences of n removed from the list
;; (drop 'a '(x y a b a c)) returns (x y b 1c)
(cond
  ((null l) nil)
  ((equal n (car l)) (drop-all n (cdr l)))
  (t (cons (car l) (drop-all n (cdr l))))))

;; **** End of function definitions ****
;;
(dribble 'fname)
(print "Testing the function: Length")

(pprint (get-lambda-expression (function length))) ;; print the function
(print 'length (a b c d))
(print (length (a b c d)))
(print '(length nil))
(print (length nil))
;; ******************************************************************************
;; Testing drop
(print "Testing the drop function")

(pprint (get-lambda-expression (function drop)))

(print '(drop a (x y a b c a))) ;; test line
(print (drop 'a '(x y a b c a))) ;; actual test

(print '(drop a (x y z)))
(print (drop 'a '(x y z)))

;; ******************************************************************************
;; Testing drop-all
;;..........................

(dribble)
THE DO STATEMENT FOR ITERATION

;; Iterative version of finding the length of a list
(defun do-length (l)
  ;; This function finds the length of a list using iteration
  ;; The DO function performs the iteration.
  (do
    ;; define the variables
    (ind1 l (cdr ind1)) ;; first variable
    (sum 0 (+ 1 sum)) ;; second variable
    ) ;; end of variable definitions
  ;; exit clause is next
  ((null ind1) sum) ;; close exit clause
  ;; body of Do statement is next
  ) ;; end of do statement
  ) ;; end of defun

;; Iterative version of finding length of a list of lists
(defun do-len2 (l)
  ;; This function finds the length of a list using iteration
  ;; The DO function performs the iteration.
  (do
    ;; define the variables
    (ind1 l (cdr ind1)) ;; first variable
    (sum 0 (cond
      ((listp (car ind1)) (+ (do-len2 (car ind1) ) sum))
      (t (+ 1 sum)))
    ) ;; end of variable declarations
    ;; exit clause is next
    ((null ind1) sum) ;; close exit clause
    ;; body of Do statement is next
    ) ;; end of do statement
  ) ;; end of defun
(defun compute (fn)
;; compute takes a function name as an argument
;; and makes a computation using the argument.
;; The lisp statements LET READ and FUNCALL are
;; introduced.
   (print '(please input two values to compute))
;;
;; (do
;;   ((num 0 (+ 1 num))
;;    ;; end of variable declarations
;;    ((> num 5))

;; body of do
   (terpri) ;; blank out a line

;; close do

;;
;; (let
;;   ((x (read))
;;    (y (read))
;;    ;; body of let
;;    (funcall fn x y))
);; close defun
PROPERTY LISTS

;; This is an example using property lists
;;
;;
;; object  property      value
;;  ----------------------
;;  animal  color       brown
;;  animal  age         find-age function
;;  dog     ako         animal
;;  fido    isa         dog
;;  fido    color       black
;;  butch   isa         dog
;;  butch   color       white
;;  penny   isa         dog
;;
;; Declare the semantic network above using property lists
;;
;; (setf (get 'animal 'color) 'brown)
;; (setf (get 'animal 'age) (function find-age))
;; (setf (get 'dog 'ako) 'animal)
;; (setf (get 'fido 'isa) 'dog)
;; (setf (get 'fido 'color) 'black)
;; (setf (get 'butch 'color) 'white)
;; (setf (get 'butch 'isa) 'dog)
;; (setf (get 'penny 'isa) 'dog)
;;
;; Implement a function to traverse the hierarchy following
;; AKO and ISA links

(defun select (obj prop)
  (cond
   ((get obj prop) (get obj prop))
   ((get obj 'isa) (select (get obj 'isa) prop))
   ((get obj 'ako) (select (get obj 'ako) prop))
   (t nil)))

;; Implement functions as values of properties
(defun find-age (date)
  (- *year* date))
(setf (get 'animal 'age) (function find-age))

(setf (get 'penny 'yearofbirth) 2000)

(funcall (get 'object 'age) (get 'penny 'yearofbirth))

;;;;
(defun selectnew (obj prop)
  (cond
   ((get obj prop) (get obj prop))
   ((get obj 'isa) (select2 (get obj 'isa) prop obj))
   ((get obj 'ako) (select2 (get obj 'ako) prop obj))
   (t nil)))

(defun select2 (obj prop original)
  (print obj)
  (print prop)
  (print original)
  (cond
   ((get obj prop)
    (cond
     ((fboundp (get obj prop)) ;; do I have a function to execute?
      (select3 (get obj prop) prop original))
     (t (get obj prop))))
   ;; value not found, use inheritance
   ((get obj 'isa) (select2 (get obj 'isa) prop original))
   ((get obj 'ako) (select2 (get obj 'ako) prop original))
   (t nil)))

;;;;
(defun select3 (foo prop original)
  (print prop)
  (print original)
  (cond
   ((equal prop 'age) (funcall foo (get original 'yearofbirth)))
   (t 'something-is-wrong))
MACROS

;; Macros
;; A macro is essentially a function that generates Lisp code.
;; The difference between a macro and a function is that a function produces results
whereas
;; a macro produces expressions - which, when evaluated, produce results.

;; Define a simple macro

(defun pos-x (l)
   `(cadr ,l))

;; Look at the macro expansion

(print (macroexpand-1 '(pos-x (a b c d))))

;; Call the macro

(print (pos-x '(a b c d)))

;; Example #2 -->

(defun iff (Test Then &optional Else)
   ;; A replacement for cond
   ;; If the first argument evaluates to non-Nil, then
   ;; evaluate and return the second argument. Otherwise,
   ;; evaluate and return the third argument (which defaults to nil

   `(cond
      ,(test ,Then)
      (t     ,Else)))

;; Expand the macro

(print (macroexpand-1 '(iff (> 4 3) (print 'greater))))

;; Use the macro

(defun testmacro (l)
   (iff (null l) nil (print l)))

(print (testmacro '(a b c d)))

(print (testmacro '() ))

;; End of macro code
;;; create an object-oriented environment for gum-ball machines
;;;
;;; (defun gum-machine (supply)
;;;   ;; this function returns a gumball machine
;;;   ;; supply represents the gumballs contained in the gumball machine
;;;   (function
;;;     (lambda ()
;;;       (prog1 (car supply)
;;;           (setq supply (cdr supply))))))

;;; Now, create a gumball machine

(defvar *hyvee-machine* (gum-machine '(red white blue green red black blue)))

;;; define a function that returns a queue object using the labels function

(defun make-queue ()
  (let ((queue nil))
    ;; body of let statement
    (labels
      (
        (empty-queue ()
          (null queue)) ;;

        (add-queue (v)
          (setf queue (append queue (list v)))) ;;

        (dequeue ()
          (cond
            ((null queue) 'empty-queue)
            (t (prog1 (car queue) (setq queue (cdr queue))))))

        (p-queue ()
          queue)

        (dispatch (m)
          (cond
            ((equal m 'empty?) #empty-queue))
    ))
  ))

;; body of labels returns dispatch function
;; since it intercepts all messages

  #'dispatch
  )  ;; close labels
  )  ;; close let
  )  ;; close defun

;;;;;;;;;;;;;;;;;;;
(defvar *myqueue* (make-queue))  ;; returns a queue object

;; check if the queue is empty
(funcall (funcall *myqueue* 'empty?))

(funcall *myqueue* '(add 10))
(funcall *myqueue* '(add 5))
(funcall *myqueue* 'dequeue)
(funcall *myqueue* 'printq)
USING DEFSTRUCT

;; Define the structure for a binary sort tree

(defun add-tree (root v)
  ;; Insert one item into the binary tree
  (let ((x (node-val root)))
    ;; body of let
    (cond
      ((> v x)
       (cond
        ((equal (node-right root) nil)
         (setf (node-right root) (make-node :val v)))
        (t (add-tree (node-right root) v))))
    
    ;; Must be less than or equal to
    
    (t
     (cond
      ((equal (node-left root) nil)
       (setf (node-left root) (make-node :val v)))
      (t
       (add-tree (node-left root) v)))))))

;; Define a depth-first tree traversal

(defun df (root v)
  ;; do a depth-first search looking for v at node root
  (cond
    ((null root) nil)
    ((equal v (node-val root)) 'found)
    ((df (node-left root) v))
    (t (df (node-right root) v))))