Quick Guide to Lisp

Program Structure:

Collection of function definitions, each of the form:

```
(defun <name> (<paramenters>)
    <function body>)
```

example:

```
(defun second (lst)
    (car (cdr lst)))

(defun third (lst)
    (car (cdr (cdr list))))
```

Comments

Anything after a semicolon ; is a comment

Built-in functions

```
(quote x)           ; do not evaluate x
    [or 'x]         ;
(car lst)           ;  returns the first element of lst
                    ;     (car '(a b c))    => a,  (car '((b c) d))  => (b c)
(cdr lst)           ; returns lst with the first element removed
                    ;     (cdr '(a b c))    => (b c),  (cdr '((b c) d) => (d)
(cons elem lst)     ; returns lst with elem inserted in the front
                    ;     (cons 'a '(b c)) => (a b c),  (cons '(a b) '(c d)) => ((a b) c d)
(list a b c)        ; returns the list (a b c).  Can take a variable number of input parameters
                    ;     (list 3 4 3 2) => (3 4 3 2),  (list 'a 'b '(c d)) => (a b (c d))
```

Predicate functions

the empty list [() or nil] represents false, and t represents true

(although, much like in C, anything other than the empty list is considered "true" in Boolean expressions)

```
(atom x)            ; returns t if x is an atom, and nil otherwise
(numberp x)         ; returns t if x is a number, and nil otherwise
(null x)            ; returns t if x is equal to nil, and nil otherwise
(and x y)           ; logical and of x and y
(or x y)            ; logical or of x and y
(not x)             ; logical negation of x
(equal x y)         ; returns t if x and y are the same atom (or a list of all the same atoms, or
                    ;     so on, recursively)
                    ;     (equal '(a (b c)) '(a (b c)) => t,  (equal '(a (b d)) '(a (b c)) => nil
```

Arithmetic functions

```
(+ x y)             ; returns the sum of x and y (can have multiple arguments)
                    ;  (- 2 3) => -1, (- 3 4 5) => -6
(- x y)             ; returns the difference of x and y (can have multiple arguments)
                    ;  (+ 2 3) => 5, (+ 3 4 5) => 12
(* x y)             ; returns the product of x and y (can have multiple arguments)
                    ;  (* 3 4 5) => 60
(/ x y)             ; returns the quotient of x and y (result is a rational number)
                    ;  (/ 4 8) => 1/2,  (/ 9 6 5) => 3/10
(rem x y)           ; returns the remainder of x/y
```

Control flow functions:

```
(cond (<test1> <expr1>)
      (<test2> <expr2>)
          ...
      (<testn> <exprn>)
      (t       <default expr>))                ; the t clause is optional
```

If <test1> evaluates to true, then return the value of <expr1>
else, if <test2> evaluates to true, then return the value of <expr2>
else, if <test3> evaluates to true, then return the value of <expr3>
...
else, if <testn> evaluates to true, then return the value of <testn>
else return the value of <exprn1>

example:

```
(cond ((equal a 0) 0)
      ((< a 0) (- 0 a))
      (t  a))
```

Loading Programs

(load "filename")

example:

(load "test.lsp")

Exiting Interpreter

(exit)

Example Functions

```
; input: non-negative integer x
; output: x!
    (defun factorial (x)
        (cond    ((equal x 0) 1)
                 (t (* x (factorial (- x 1))))))

; input: list lst
; output: length of lst
    (defun length (lst)
        (cond    ((null lst) 0)
                 (t (+ 1 (length (cdr lst))))))

; input: list lst and element elem
; output: t iff elem is a top-level member of lst
    (defun member (elem lst)
        (cond    ((null lst) nil)
                 ((equal elem (car lst)) t)
                 (t member elem (cdr lst))))
```

```
; input: list lst and element elem
; output: number of times elem occurs as a top-level element of lst
    (defun count-member (elem lst)
        (cond    ((null lst) 0)
                 ((equal (car lst) elem) (+ 1 (count-member elem (cdr lst))))
                 (t    (count-member elem (cdr lst)))))

; input: list lst and number n
; output: the nth element of lst
    (defun nth (lst n)
        (cond    ((equal n 1) (car lst))
                 (t (nth (cdr lst) (- n 1)))))

; input: list lst
; output: total number of atoms contained in lst
    (defun count-atoms (lst)
        (cond    ((null lst) 0)
                 ((atom (car lst)) (+ 1 (count-atoms (cdr lst))))
                 (t (+ (count-atoms (car lst)) (count-atoms (cdr lst))))))

; input: list lst
; output: total number of atoms contained in lst
    (defun count-atoms2 (lst)
        (cond    ((null lst) 0)
                 ((atom lst) 1)
                 (t (+ (count-atoms2 (car lst)) (count-atoms2 (cdr lst))))))

; input: lists lst1 and lst2
; output: lst2 appended to the end of lst1
    (defun append (lst1 lst2)
        (cond    ((null lst1) lst2)
                 (t (cons (car lst1) (append (cdr lst1) lst2)))))

; input: element elem and list lst
; output: lst with elem inserted as last element
    (defun cons-to-end (elem lst)
        (cond    ((null lst) (cons elem lst))
                 (t (cons (car lst) (cons-to-end elem (cdr lst))))))

; input: element elem and list lst
; output: lst with elem inserted as last element
    (defun cons-to-end2 (elem lst)
        (append lst (list elem)))

; input: element elem and list lst
; output: lst with elem inserted as last element
    (defun cons-to-end3 (elem lst)
        (append list (cons elem nil))

; input: element elem and list lst
; output: lst with first top-level occurrence of elem removed
    (defun remove-first (elem lst)
        (cond    ((null lst) lst)
                 ((equal elem (car lst)) (cdr lst))
                 (t (cons (car lst) (remove-first elem (cdr lst))))))
```

```
; input: element elem and list lst
; output: lst with all top-level occurrences of elem removed
      (defun remove (elem lst)
            (cond     ((null lst) lst)
                      ((equal elem (car lst)) (remove elem (cdr lst)))
                      (t (cons (car lst) (remove elem (cdr lst))))))


; input: element elem and list lst
; output: lst with all occurrences of elem removed
      (defun remove-all (elem lst)
            (cond     ((null lst) nil)
                      ((atom lst) lst)
                      ((equal (car lst) elem) (remove-all elem (cdr lst)))
                      (t (cons  (remove-all elem (car lst))
                                (remove-all elem (cdr lst))))))

; input: list lst
; output: lst reversed
      (defun reverse (lst)
            (cond     ((null lst) lst)
                      (t (cons-to-end (car lst) (reverse (cdr lst))))))
```