Lisp syntax sheet: quiz 1

```
; comments (single line, to right of the semicolon)

; global variable, constant declaration/initialization
(defvar x 10)
(defconstant pi 3.14)

; simple data types and syntax:
;    real numbers, e.g. 5.123
;    integers, e.g. 1024
;    rationals, e.g. 14/3
;    strings, e.g. "foo"
;    characters, e.g. #\c
;    booleans, e.g. t and nil

; prefix style function calls and operations, e.g.
;       (+ (* x y) (- (/ a b) c))
; equivalent of C++ (x*y) + ((a/b) - c)
;
; common math functions:
;    +, -, *, /, tan sin cos sqrt mod ceiling floor expt
; e.g. (expt 2 3) is 2 to the power of 3

; typechecking functions return t (true) if the parameter
;    has the specified type, nil (false) otherwise
; (stringp x) (integerp x) (realp x) (numberp x) (listp x)
; (characterp x) (symbolp x) (functionp x) etc

; function to read and return next item of user input
; (can be number, string, character, list, etc)
(read)

; format can display text, variable, newline (~%), returns nil
(format t "Value in x is ~A~%" x)

; if/else statement to return larger of x and y
;    (skips error checking for numbers in this example)
(if (> x y) x y)

; cond statements take a list of pairs, each has
;    a condition to check and value to return if true
;    t used as final pair's condition since it is always true
(cond
    ((stringp z) (format t "length of string ~A is ~A~%" z (length z)))
    ((listp z) (format t "length of list ~A is ~A~%" z (length z)))
    ((numberp z) (format t "given number, value is ~A~%" z))
    (t (format t "given unexpected value: ~A~%" z)))

; numeric comparison functions: < <= > >= = /=
; string comparisons: string= string> etc
; boolean operators: and or not

; list syntax '(1 2 3 4) where elements can be any type
; empty list denoted nil or '()

; sequence functions (work on lists)
(length L)     ; returns length of list L
(member e L)   ; true if e is element of L, false otherwise
(count e L)    ; counts how often e is in L
(position e L) ; finds first position of e in L
```

```
; list functions
(list x y z)   ; returns list with elements x, y, z
(null L)       ; true iff L is empty list
(car L)        ; returns first element of L
(cdr L)        ; returns list of all the elements in L except the first
(cons e L)     ; return a list with element e followed by elements of L
(reverse L)    ; returned reversed copy of L
(append L1 L2) ; return list with elements of L1 then elements of L2

; sample function definition with doc string and cond
(defun biggerSquare (x y)
   "returns square of larger of numbers x, y, nil on error"
   (cond
      ; error if neither is a number
      ((and (not (numberp x)) (not (numberp y))) nil)

      ; if exactly one is a number then return it
      ((not (numberp x)) y)
      ((not (numberp y)) x)

      ; if x is larger return its square
      ((> x y) (* x x))

      ; otherwise return y's square
      (t (* y y))))

; sample function with let block for local variable
;    note last value/statement in a let block is what it returns,
;    and last value/statement in a function is what it returns
; (and here the let block is the last statement in the function)
(defun getValSums (N)
   "get 3 values from user, store in variables, display, return last"
   (let ((first nil) (second nil) (third nil))
      ; get the three values
      (format t "Enter your first value: ")
      (setf first (read))
      (format t "Enter your first value: ")
      (setf second (read))
      (format t "Enter your first value: ")
      (setf third (read))
      ; display them all
      (format t "Values: ~A, ~A, ~A~%" first second third)
      ; return the last one
      third))

; sample recursion through list
(defun sumElements (L)
   "display sum of numbers in a list"
   (cond
       ; error check
       ((not (listp L)) (format t "Error: not a list (~A)~%" L))
       ; empty list has sum 0
       ((null L) 0)
       ; ignore current front element if not a number,
       ;    call recursively on rest of list
       ((not (numberp (car L))) (sumElements (cdr L)))
       ; return sum of front element and rest of list
       (t (+ (car L) (cdr L)))))
```