Midterm Examination, Cmput 325

LAST NAME ___________________________________

FIRST NAME ___________________________________

INSTRUCTIONS:

- In writing Lisp programs, you may use any built-in functions that have been allowed in the first two assignments. Besides arithmetic and comparison functions, some of these functions are listed below

```
(atom x)
(null x)
(eq x y)
(equal x y)
(numberp x)
(append x y)
(car x)
(cdr x)
(cons x y)
(if x y z)
(cond  ... )
(let ((x y) ...(u v)) z)
(defun ...)
(quote x) and its short form 'x
(mapcar x y)
(reduce x y)
(lambda ...)
(funcall ...)
(apply ...)
(list ...)
......
and any combination of car and cdr, such as (cadr ...), (cdaar ...), etc.
```

[10 marks]
(a) Show the simplest S-expression that is stored internally by the following structure.

```
     ---------   ---------             ---------
  -> |   |   |-> |   |   |-----------> |   | / |
     ---------   ---------             ---------
       |          |                      |
       V          V                      V
     --------   ---------                d
     |   | /|   |   |   | -> c
     --------   ---------
       |          |
       V          V
       a          b
```

Your answer:

(b) When we define a function in Lisp using defun, the definition has to be stored before it can be processed. Draw the machine level representation of the following expression.

```
(defun max (X Y) (if (> X Y) X Y))
```

(c) Let's bind the above expression to an atom, say a, by the code

```
(set 'a '(defun max (X Y) (if (> X Y) X Y)))
```

Write the Lisp code that returns the subexpress `(> X Y)` of the above expression. E.g., the code `(car a)` returns the atom DEFUN.

Your answre:

[5 marks] Recall that boolean constants *true*, *false*, and operator *NOT* are defined as: $T = (\lambda xy \mid x)$, $F = (\lambda xy \mid y)$, and $NOT = (\lambda x \mid xFT)$, respectively. Simplify the following expression (show all the steps).

$$F\ T\ F\ a\ (NOT\ F\ b\ c)$$

[6 marks] Consider a boolean operator, denoted by OP, which has the following truth table.

```
X  Y        OP  X Y
--------------------
T  T          T
T  F          F
F  T          F
F  F          T
--------------------
```

Define a lambda expression for OP, and simplify it if possible. Verify that your definition works by applying it to two of the four cases specified below.

OP = $(\lambda xy \mid$ ...

```
OP F T
```

```
OP T F
```

[4 marks] Compile the following expression to SECD code.

```
(- (* 5 4) (+ 2 4))
```

[10 marks] For the lambda expression below, where we draw underlines to help you identify its components, evaluate it by the interpreter based on context and closure by answering the corresponding questions. The initial context is assumed to be CT0 = $\{z \to 4\}$. Clearly indicate your answer. You don't need to show how you get your answers. Assume the evaluation starts from

```
eval
    ((lambda (f x y) (f (f x y) z)) (lambda (w v) (+ w v)) 3 5)
     ------------------------------ ---------------------- - -
in CT0
```

Show the context when `(f (f x y) z)` is evaluated; i.e.,

```
eval (f (f x y) z) in __________________________________________
```

Show the context when `(f x y)` is evaluated; i.e.,

```
eval (f x y) in __________________________________________
```

The expression `(+ w v)` will be evaluated twice, i.e., the interpreter will make recursive calls, twice with `(+ w v)` as the expression to be evaluated. Show these contexts.

```
In the first call:

     eval (+ w v) in __________________________________________

In the second call:

     eval (+ w v) in __________________________________________
```

[12 marks] Consider the following lisp program.

```
(defun f (L1 L2)
    (if (null L1)
        (let ((s (g L2))) (- 0 s))
        (+ (car L1) (f (cdr L1) L2))))

(defun g (L)
    (cond
        ((null L) 0)
        ((null (cdr L)) (car L))
        (t (+ (cadr L) (g (cddr L))))))
```

Show the result of evaluating each expression below.

```
(g '(1 2 3))                       your answer:


(g '(1 2 3 4))                     your answer:


(f '(1 2 3) '(2 4))                your answer:


(f '(5 3 6) '(1 4 2))              your answer:
```

[6 marks] Consider the following lisp program.

```
(defun h (L)
   (cond
       ((null L) T)
       ((atom L) (not L))
       (t (h (h (cdr L))))))
```

Show the result of evaluating each expression below.

```
(h '(a b c d))                 your answer:


(h '((a b) (b (e)) d))         your answer:
```

[6 marks] Suppose we have the following definitions in Lisp.

```
(defun filter (p L)
         (if (null L)
             nil
             (if (funcall p (car L))
                 (cons (car L) (filter p (cdr L)))
                 (filter p (cdr L)))))
(defun g5 (x) (> x 5))
(defun c3 (x) (if (> x 3) (+ x 1) (- x 1)))
```

Show the result of evaluating each expression below.

```
        (filter 'g5 '(7 2 8 4 5 6))

  your answer:

        (mapcar 'c3  (filter 'numberp '(a a b 3 8 d 6)))

  your answer:
```

[6 marks] Write a Lisp function `(defun complement (S1 S2) ...)`, where `S1` and `S2` are lists of atoms such that `S2` is a subset of `S1` and the function returns the list of those atoms that are in `S1` but not in `S2`. The order of the resulting list is unimportant. E.g.,

```
(complement '(a b c d e) '(a c)) ==> (b d e)
```

Assume that neither `S1` nor `S2` contains duplicated elements. If you need a membership function, you should define it.

[6 marks] Define a Lisp function `(defun rotate (L) ...)` that moves the first element of L to the end of the list and every other element to its left. Note that although an element of a list may be an atom or a sublist, no sublist should be rotated. In the case where L is an empty list, NIL should be returned. E.g.

```
(rotate '(a b (1 2) (c d)))  ==> (b (1 2) (c d) a)
```

[9 marks] Extend the above function so that every sublist is also rotated. Call the resulting function `rotateAll`. For example,

```
(rotateAll '(a b (1 2 3 4) (c d)))     ==> (b (2 3 4 1) (d c) a)
(rotateAll '((a b) () (1 2 3 4) c d))  ==> (nil (2 3 4 1) c d (b a))
(rotateAll '(a b nil (1 2 (3 4))))     ==> (b nil (2 (4 3) 1) a)
```