York University

Department of Computer Science & Engineering

Midterm Exam

CSE3401 Summer 2009

July 15th, 2009

Student Name: ..... Student #: .....

1. The time is approximately 80 minutes.
2. This is a closed book exam.
   No examination aids are permitted.
3. All programming is to include comments.
   When using functions like MAPCAR, APPLY etc. clearly indicate what is the effect you want using diagrams and annotations.
4. Use the reverse of the pages for additional space. Cross your temporary work off.

Q#	Points	Score
1	10	_____
2	10	_____
3	14	_____
4	12	_____
5	10	_____
6	10	_____
Total	66	_____

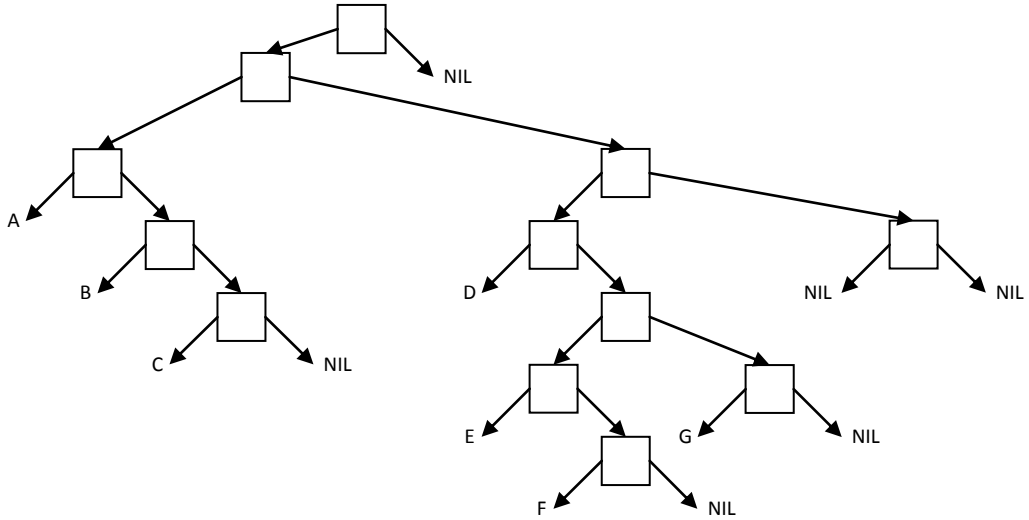
Question 1 (10 points)

a) Write the following s-expression in list notation as fully dotted notation and draw the representation. Use the reverse of the page if more space required for the drawing.

(((A B C) (D (E F) G) ()))

Answer:

(((A . (B . (C . NIL))) . ((D . ((E . (F . NIL)) . (G . NIL))) . (NIL . NIL))))



b) Simplify the following fully-dotted s-expression to list notation as much as possible and show the steps.

((A . (B . (C . NIL))) . (((D . (E . NIL)) . F) . (G . NIL)))

Answer:

((A . (B . (C))) . (((D . (E)) . F) . (G)))
((A . (B C)) . (((D E) . F) G))
((A B C) . (((D E) . F) G))
((A B C)((D E) . F) G)

Question 2 (10 points + 2 bonus points)

Assume the following have been executed in the LISP interpreter in sequence:
(setq a 'x) → X
(setq b 'y) → Y
(set b 'a) → A
(setq x ((lambda (n) (cons n 'x)) 'b)) → (B . X)
(defun a (y) (cons y '(y))) → A
(setq z 'd) → D
(setq d 'e) → E
(setq e y) → A
(setq ls '((1 2 3 4)(5 6 7 8))) → ((1 2 3 4)(5 6 7 8))

What are the outputs of the evaluation of the following forms?
(cons y 'c)
→ (A . C)
(cdr (a b))
→ (Y)
(list a b)
→ (X Y)
(append '(a b) (a b))
→ (A B Y Y)
(eval z)
→ E
(caddar ls)
→ 3
(apply 'mapcar 'list ls)
→ ((1 5) (2 6) (3 7) (4 8))
(funcall 'mapcar 'list ls)
→ (((1 2 3 4)) ((5 6 7 8)))
(apply 'mapcar '+ ls)
→ (6 8 10 12)
(mapcar 'reverse ls)
→ ((4 3 2 1) (8 7 6 5))

BONUS
(eval (eval (eval (eval z))))
→ (B . X)
(mapcar (bu (rev 'cons) 'x) '(1 2 3 4 5))
→ ((1 . X) (2 . X) (3 . X) (4 . X) (5 . X))

Question 3 (14 point + 3 bonus points)

a) What is functional programming and what are its main attributes?

Functional programming consists of writing functions that have functions as input and frequently as output. That is writing functions that themselves create new functions. Use of generalized functions that abstract control flow patterns -- e.g. mapcar and reduce. Functional programs have no explicit loops (recursion), have no sequencing at a low level, have no local variables. Frequently input is a single list of parameters.
Characteristics of functional programming: Meaningful Units of Work, Requires Abstraction (ex. abstract out the control flow patterns).

b) What are mapping functions and what abstraction do they represent?

Mapping functions are functions that can apply a function repeatedly to any number of sets of arguments. Please note that they take functions as arguments.
Examples of mapping functions are MAPCAR, MAPC, MAPLIST, MAPL

c) What is a lambda expression in LISP and when is it useful?

We can have the function with no name using the LAMBDA function and by constructing lambda expressions. Lambda expressions allow defining functions without the need to permanently storing it (i.e. associating it with a name). Lambda notation allows for lambda abstraction, which is to abstract what the function does regardless of the name given to it.

d) What are the elements of successful recursion?

Three elements of successful recursive solution:
1) Recursive Relation i.e. explain why and how the function needs to invoke itself
2) Grounding Condition i.e. when the function should stop invoking itself
3) Progress towards the goal i.e. make sure that if the function invokes itself then the parameter passed to the recursive invocation is closer to the grounding condition i.e. the function invokes itself with a smaller and smaller argument if the grounding condition is an empty list.

e) How are logical values represented and handled in LISP?

Nil represents false, any non-nil value represents TRUE. For convenience atom t is done as a special symbol – its value cannot be changed, it is used to clearly represent TRUE and it evaluates to t (self).

f) What are special symbols and what is special about NIL?

Values of special symbols cannot be changed and they evaluate to themselves. Nil represents false, any non-nil value represents TRUE. Atom nil is special its value cannot be changed, it is used to clearly represent FALSE and it evaluates to nil (self).

g) Why are destructive list manipulation functions powerful, dangerous and efficient? Give an example of destructive list manipulation.
There are functions that will manipulate the memory directly rather than build a new object and return it. They are dangerous and power full because they allow building any graph structures including circular ones. The destructive list manipulation functions are very powerful (you can do things that no other functions can do) and very efficient.
RPLACA – substitutes for the left pointer of a cons cell
RPLACD – substitutes for the right pointer of a cons cell
NCONC – glues two lists destructively
APPEND – creates a new list and by copying and glueing the given lists

h) **[BONUS]** Both (reduce '+ '(1 2 3 45)) and (apply '+ '(1 2 3 45)) work and produce the same result. What is the difference between REDUCE and APPLY? Give an example when they behave differently on the same input.
REDUCE takes any function that is binary (takes 2 arguments, it includes flexible number of arguments functions) even if the list of arguments is longer than 2. APPLY works only if the number of arguments of the function that it takes is the same as the length of the list of arguments .Flexible argument functions will work the same in both.
(reduce 'cons '(1 2 3 45)) → (((1 . 2) . 3) . 45)
(apply 'cons '(1 2 3 45)) → error

Question 4 (12 points + 3 bonus points)

a) You are given a matrix ((a b c) (d e f)) that represents a set linear equations
a * x + b * y = c
d * x + e * y = f
Write a LISP function that will output the pair (x y) that satisfies the two equations.

```
(defun a (ls) (caar ls))      (defun b (ls) (cadar ls))      (defun c (ls) (caddar ls))
(defun d (ls) (caadr ls))     (defun e (ls) (cadadr ls))     (defun f (ls) (caddadr ls))
(defun solve (ls)
  (list
    (/ (- (* (c ls) (e ls)) (* (b ls) (f ls))) (- (* (a ls) (e ls)) (* (b ls) (d ls))))
    (/ (- (* (c ls) (d ls)) (* (a ls) (f ls))) (- (* (b ls) (d ls)) (* (a ls) (e ls))))
  ))
```

b) Write a recursive LISP function that will output the last element of a given list.

```
(defun last-element (ls)
  (cond
     ((null ls) nil)
     ((null (cdr ls)) (car ls))
     (t (last-element (cdr ls)))
  ))
```

c) Write a macro EUCLID that takes arguments x and y and expands to (sqrt (+ (* x x) (* y y)))

```
(defmacro Euclid (x y)
  (list 'sqrt (list '+ (list '* x x) (list '* y y))))
```

d) **[BONUS]** Write a LISP functional, that uses a lambda expression and mapping functions that will output the last element of a given list.

```
(defun f-last-element (ls)
  (reduce 'append
    (maplist
      #'(lambda (x) (and (not (null x)) (null (cdr x)) (car x)))
      ls
    )))
```

Question 5 (10 points)

We are given a list with possibly nested sublists to any level. The list contains only atoms GREEN and RED. Write a LISP functional that will "repaint" any given described input list to another list with atoms GREEN and RED as follows: if an atom is the same as the next atom or there is no next element on the list then replace it with RED otherwise replace it with GREEN. For example:
(RED RED (RED GREEN GREEN ()) RED) will produce (RED GREEN (GREEN RED GREEN ()) RED)

```
(defun repaint (ls)
  (if (null ls)
    nil
    (cons
      (cond
        ((listp (car ls)) (repaint (car ls)))
        ((or (null (cdr ls)) (equal (car ls) (cadr ls))) 'red)
        (t 'green)
      )
      (repaint (cdr ls))
    )
  )
)
```

Question 6 (10 points)

NOTE: Try to make your code as close to executable as possible, use helper functions and clearly explain your logic.

a) Write a LISP recursive function that will output every second element of a given list.

```
(defun every-second (ls)
  (if (null (cdr ls))
    nil
    (cons
      (cadr ls)
      (every-second (cddr ls))
    )
  )
)
```

b) Write a LISP functional BIGGEST-SUBLIST that will measure the maximum length of the bottom sublists of a given list that contains sublists to any level deep.
TIP: Think of what was done in Assignment #2 Question #4 and if any of your work could be adopted here. Assume you have the function (max x y) that returns the bigger of two numbers x and y.

```
(defun biggest-sublist (ls)
  (reduce 'max (mapcar 'helper ls)))

(defun helper (obj)
  (cond
    ((atom obj) 0)
    ((atom (car obj)) (length obj))
    (t (biggest-sublist obj))
  ))
```