Family name _____SOLUTION______

Given name(s) ______________________

Student number ______________________

York University
Faculty of Science & Engineering / Faculty of Arts
Department of Computer Science & Engineering

Class Test 1

AK/AS/SC/COSC3401.03
Functional & Logic Programming

2006 June 8

Instructions

1. The test time is approximately 90 minutes.
2. This is a closed book examination. No examination aids are permitted.
3. All questions are to be attempted.
4. Using annotated diagrams, examples, complete sentences and paragraphs will increase the effectiveness of your answer.
5. All programming is to include comments
6. If a question is ambiguous or unclear then please write your assumptions and proceed to answer the question.

Ques	Max	Mark
1	9	_____
2	2	_____
3	4	_____
4	6	_____
5	7	_____
6	4	_____
7	10	_____
Total		_____
Letter grade		_____

Question 1 (9 marks)

Assume the following forms have been typed into the interpreter and evaluated.

```
( defun  a  ( y )  ( list  y  'y ) )

( setq  a  'c )

( defun  b ( x )  ( maplist  #'list  x ) )

(setq  c  'd )

( setq  e  ( (lambda  ( x )  x )  'a ) )

( setq  b  'e )

 ( set  c  'b )

( setq  f  '(  a  ( b ) ) )

( setq  g  ( cons  a  ( cdr  f ) ) )
```

What will the following forms evaluate to?

1. (eval a) ______D______
2. (funcall e ‘c) _____(C Y)____
3. (b f) (((A (B))) (((B))))
4. (cons a   b) ____(C . E)____
5. (member a g) ____(C (B))____
6. (eval (eval (eval (eval (quote e))))) _______B_______
7. (apply (caddr g)   (cdr g)) _____(((B)))_____
8. (mapcar #’list g) __((C) ((B)))_____
9. (eq (cdr g) (list (cadr f))) ______NIL_______

Question 2 (2 marks)

Two of the following s-expressions are equivalent. Which two?

1. (a b . (c . d))
2. (a (b . c) . d)
3. (a ((b . c) . d))
4. (a . ((b . c) . d))

Answer: 2 and 4

Question 3 (4 marks)

Explain why recursive programming techniques in Lisp require that COND and IF**not** be functions.

Recursive programs must use conditional expressions since there has to be a non-recursive as well as a recursive branch. But if IF were a function then the forms on both branches would be evaluated before a branch was executed, causing an infinite loop by evaluating the recursive branch each time. The same applies to the COND since it is a macro which expands to nested IFs.

Question 4 (6 marks)

Define a recursive Lisp function **remove** that takes two arguments, a list of atoms X and a list Y and removes all elements of X from Y. The function returns a list that contains only those elements in Y that do not appear in X. The elements in the result should be in the same order as they are in Y. For example:

```
(remove '(a b c) '(d e a b a c e f))                  →   (d e e f)
(remove '(a b c) '(d e (a b) a (c) e f))              →   (d e nil nil e f)
(remove '(a b c) '(d e (a (a d a b) (b)) a (c) e f))  →   (d e ((d) nil) nil e f)
```

You may use **ONLY** the Lisp functions defun, cond, car, cdr, cons, append, list, null, member, atom, listp. You may use combinations of car and cdr such as caddr. You may define a helper function.

```
; remove takes two arguments X and Y where X is a list of atoms
; the function removes all elements in X from Y and returns the resulting list
; this solution recurses on Y, another solution is to recurse on X and use a helper function.

(defun remove (X  Y)
  (cond (( null Y ) nil )
        (( member  (car  Y)  X) (remove X (cdr Y)))
        (( atom ( car Y)) (cons  (car  Y) (remove X  (cdr  Y))))
        ( t (cons (remove  X  (car Y)) (remove  X  (cdr Y))))))
```

Question 5 (7 marks)

Define a recursive Lisp function **longestRun** that takes a list of atoms and returns the length of the longest run of consecutive occurrences of the same atom in the list. For example,

```
(longestRun  '( b  a  b  a  b  b  b  a ) )        → 3
(longestRun  '( a  b  a  a  c  b ) )              → 2
(longestRun  '(c) )                               → 1
(longestRun  '( ) )                               → 0
```

You may use **ONLY** the Lisp functions defun, cond, car, cdr, cons, append, null, <, >, equal, max, +. You may use combinations of car and cdr such as caddr. You may define a helper function.

```
; longest run takes one argument, a list, and returns the length ; of the longest run of consecutive
; occurrences of the same atom in the list
; precondition: list is a valid list

(defun longestRun (list)
        (cond ( ( null list ) 0 )
                ( t (longestHelp (car list ) (cdr list ) 1 0 ) )
         )
)

; longesthelp takes 4 arguments, the current element being looked at, the list, the current count
; for element elem and the maximum count so far
; helper function, all the work is done here

(defun longestHelp (elem list c1 c2)
        (cond ( ( null list )
                        (max c1  c2 ) )        ; empty list return either current max or max so far
        ((equal elem (car list))
                (longestHelp elem (cdr list) (+ 1 c1) c2 ) )      ; next element is same as elem
        ( t ( longestHelp (car list) (cdr list) 1 (max c1 c2)))      ; next element is diff than elem
        )
)
```

Question 6 (4 marks)

What is functional programming? What are the prime attributes of functional programs?

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 low level, have no local variables. Frequently input is a single list of parameters.

Question 7 (10 marks)

For the following questions you may use **ONLY** the Lisp functions defun, cond, car, cdr, cons, append, list, null, numberp, atom, listp, mapcar, maplist, reduce. You may use combinations of car and cdr such as caddr.

During the test + and * were added to the above list.

For the following questions **DO NOT** use helper functions but you may use lambda expressions.

(a) Write a functional program (no explicit recursion), **easytype**, that given an input list, returns a corresponding list of num, atom or list depending on whether an item at the top level is a number, atom or list. For example:

(easytype '(a 3 b (b) 2 nil)) → (atom num atom list num atom)

```
; easytype determines whether the top-level elements of a list are numbers, atoms or
; lists and creates a new list indicating this
; note:  the order of the cond is important since numbers are both numbers and
; atoms and nil is both an atom and a list.

(defun easytype (alist)
      (mapcar #'(lambda (x)  (cond (( numberp  x) 'num )
                                   (( atom x ) 'atom )
                                   ( t 'list )))  alist ))
```

(b) Write a functional program (no explicit recursion), **salary** that computes a list of pairs (name payment), where each payment is computed by multiplying hours and rate for that person. Employees should appear in the same order in the output as in the input. For example:

(salary '((lee 25 10) (sam 40 8) (ann 10 20))) → ((lee 250) (sam 320) (ann 200))

```
; salary computes the salary by multiplying hours and rate for each person
; returns a list of lists where each inner list has the name of the person and the person's
; salary
(defun salary (list)
            (mapcar  #'(lambda (person) ( list  (car  person) (*  (cadr person)
                                                                   (caddr person))) list))
```

(c) Write a functional program (no explicit recursion), **totalpay** that computes the sum of all payments for the employer for a list of employees. For example:

```
(totalpay '((tom 10 20)))                          → 200
(totalpay '((lee  25  10)  (sam  40  8)  (ann  10  20)))→ 770
```

```
; the function computes the totalsalary of all people  by multiplying hours and rate  for
; each person and then summing this number
(defun totalpay (list)
        (reduce  #'+  (mapcar  #'(lambda (person) (*  (cadr person)
        (caddr person))) list)))
```