Family name _____Solution_____

Given name(s) _____

Student number _____

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

Class Test 1

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

2006 October 11

Instructions

1. The test time is approximately 75 minutes.
2. This is a closed book examination. No examination aids are permitted.
3. All questions are to be attempted.
4. All questions are of equal value.
5. Each question is evaluated on the York University letter grade scale A+, A, ..., D, E, F.
6. Using annotated diagrams, examples, complete sentences and paragraphs will increase the effectiveness of your answer.
7. All programming is to include comments. When using functions like apply, mapcar, append, etc. clearly indicate what is the effect you want using diagrams and/or symbolic notation.
8. If a question is ambiguous or unclear then please write your assumptions and proceed to answer the question.

Ques	Max	Mark
1	**9**	_____
2	**9**	_____
3	**9**	_____
4	**9**	_____
5	**9**	_____
Total		_____
Letter grade		_____

Question 1

Trace and annotate the evaluation of the following lambda expression.

```
{ □ A,B . A * B[A - 1]}[ { □ B . { □ A . B * A}[1 + B]} [ 2 ]
                       , { □ B . { □ A . B + A}} [ 7 ] ]
```

>>> Answer

To annotate means to make notes – explain what you are doing. This should be automatic with programs and with mathematics. Instruction 7 should be taken seriously in all your COSC work.

Arguments are evaluated first, as in all programming languages.

Have two arguments
1 – { □ B . { □ A . B * A}[1 + B]} [2] – is passed to A
2 – { □ B . { □ A . B + A}} [7] – is passed to B

1.1 Evaluate expression (1) – substitute 2 for B (argument is already evaluated)
{ □ A . 2 * A}[1 + 2]
1.1.1 Evaluate the argument [1 + 2] ==> 3
1.1.2 Substitute A = 3 ==> 2 * 3 ==> 6
Substitute the result for A at the outer level

Have to pass parameter before evaluating the body of a function. To evaluate body first is magic.

2.1 Evaluate expression (2) – substitute 7 for B (argument is already evaluated)
. { □ A . 7 + A}
No further evaluation is possible so substitute this for B at the outer level

At the outer level we now have

```
{ □ A,B . A + B[A - 1]}[ 6 , { □ A . 7 + A}]
```

Substitute the arguments for A and B

```
6 + { □ A . 7 + A}[6 – 1]
```

Evaluate the argument: 6 – 1 ==> 5
Substitute for A

```
6 + {7 + 5}
```

Evaluate the expression: 6 + {7 + 5} ==> 72

Not doing well on this question indicates you do not understand how programs are executed.

<<<

Question 2

You may write support functions, although you get lower evaluation if you do. The only functions you may use are the Lisp functions car and cdr and longer abbreviations, cons, cond, equal, atom, null and specific functions mentioned in each part.

A Write a recursive function, `(defun intersection (list1 list2) ... )`, that computes the set intersection of list1 and list2. Use the member function – `(member item list)`, it returns the sublist beginning at the item, if item is in the list and returns nil otherwise.

>>> Answer

```
(defun myinter (list1 list2)
 (cond ((null list1) nil)              ; More items to check in list 1?
       ((member (car list1) list2)     ; Do we want the item in the result?
          (cons (car list1)            ; Yes, keep it; Try the next item in list1.
                (myinter (cdr list1) list2)))
       (t (myinter (cdr list1) list2))        ; No, not in the result.
))
```

<<<

B Define a recursive Lisp function, `dup`, which checks whether its argument is a list containing two successive elements at the top level that are equal.

```
(dup '(A B B C)  t
(dup '(A (B) B C)  nil
```

>>> Answer

```
(defun dup (alist)
  (cond ((null (cdr alist)) nil)          ; False for lists of length 0 and 1
        ((equal (car alist) (cadr alist)) t)  ; True if equal pair
        (t (dup (cdr alist)))           ; First pair not equal
))                                      ; Try from next on list
```

<<<

Question 3

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

>>> Answer

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.

<<<

B Write a Lisp functional program, `replace(list)`, (no explicit recursion) that uses a lambda function to replace with nil every item at the top level of a list that is a list. You may not use `listp`.

Example `(replace '(1 () 2 nil 3 (a b) 4 (a (b) c) 5))`
`→ (1 nil 2 nil 3 nil 4 nil 5)`

>>> Answer

```
(defun replace (list)
 (mapcar #'(lambda (item)                ; Process every list item
             (cond ((atom item) item)    ; An atom remains as is
                   (t nil)))             ; A list is replaced
         list)
)
```

<<<

C Write a functional program, `compress(list1 list2)`, (no explicit recursion) that uses a lambda function to produces the sum of the pair-wise subtraction of the smaller numbers from larger numbers.

Example `(compress '(10 20 30 40) '(5 21 33 39))` → `10`

>>> Answer

```
   (defun compress (l1 l2)
     (reduce-pl '+                  ; Add all the substraction pairs
       (mapcar #'(lambda (a b)              ; Process an item from each list
                   (cond ((< a b) (- b a)) ; by subtracting the smaller
                         (t (- a b))))     ; from the larger
               l1 l2)
   ))
```

<<<

Question 4

A What are macros? When are they used?

```
>>> Answer
```

Macros are Lisp functions that when invoked with appropriate parameters create as output Lisp program text. Macros are used to create custom and more understandable syntax. Macros are often used in place of functions to remove function call execution time overhead. Many apparent functions in Lisp are actually macros.

```
<<<
```

B Complete the macro definition, without using backquote, of `our-if` that translates the following macro call

`(our-if a then b)` translates into `(cond (a b))`

```
(defmacro our-if   ;; complete the parameters and body
```

```
>>> Answer
```

Start with the body you want, (cond (a b)), and replace every "(" with "(list", quote the constants; leave the parameters unquoted as they need to be evaluated. The header matches the invoking sequence you want to use.

```
(defmacro our-if (theCondition then thenExpr)
   (list 'cond (list theCondition thenExpr))
)
```

```
<<<<
```

C Complete the macro definition of `our-if` using backquote.

```
(defmacro our-if   ;; complete the parameters and body
```

```
>>> Answer
```

Start with the body you want, (cond (a b)). Put a backquote in front of the S-expression and a comma in front of every parameter. The header matches the invoking sequence you want to use.

```
(defmacro our-if (theCondition then thenExpr)
   `(cond (,theCondition ,thenExpr))
)
```

```
<<<<
```

Question 5

A Assume the following functional program in Backus notation is correct. Explain step by step what the program does. The input is two lists of numbers of the same length. ^ and – are the exponentiation and subtraction operators.

```
(/ +) o (□ (bu ^ 2)) o (□ –) o trans : <<a1, a2, …, an>, <b1, b2, … bn>>
```

>>> Answer

trans – computes the transpose of a 2-d matrix.
Result is < <a1, b1>, <a2, b2>, … , <an, bn> >

(□ –) applies the minus operator to each sublist with input from the preceding function
Result is < a1–b1 , a2–b2 , … , an – bn >

(□ (bu ^ 2)) applies the function (bu ^ 2) to each item in the input list (from the preceding function)
(bu ^ 2) creates a unary function from the binary exponentiation function with the first argument fixed to the number 2 (the base of the exponent) The exponents come from the list.
Result is < 2^(a1–b1) , 2^(a2–b2) , … , 2^(an – bn) >

(/ +) reduces the input list using addition.
Result is 2^(a1–b1) + 2^(a2–b2) + … + 2^(an – bn)

<<<

B Write a program in Backus's notation that computes the arithmetic mean of a list of integers. Assume you are given the function `length` that returns the length of a list. The function divide ÷ divides two numbers.

>>> Solution

```
Arthimetic_mean ::=  ÷ o [ (/+)  , length ]
```

Explanation

```
[ (/+)  , length ]      -- produce the sum of the numbers as the first number in
                           the list and the length of the list as the second number
                           in the list

÷                       -- the division of the sum of the numbers by the number of numbers.
```

<<<

For remarking you need to write a note stating clearly and exactly where you believe your grade should be increased or decreased. Remember that the grade is a qualitative one. You need to explain why you believe the quality of your answer should, for example, if you think the grade should go up, be good (B) and not competent (C+), or, if you think the grade should go down, very good (B+) and not excellent (A).

The entire test will be reevaluated. Your grade may go up, it may stay the same, or it may go down. I will look over the entire test and see if the grades good, excellent, minimal, etc are applicable to the work as a whole (see the web page on grading in the course) independent of the points assigned to the parts.