COMP348/1 CC: Principles of Programming Languages
(Summer 2, 2009)

Midterm Exam

Professor: J. Bentahar
Date: Monday, July 27, 2009
Duration: 100 minutes

NAME: _____ **ID:** _____

INSTRUCTIONS:

- Answer all questions on these sheets in the space provided, and if you run out of space please use the back of the page.
- The use of ENCS calculators is permitted.
- This exam is 7 pages long, including the cover page. Check that your copy is complete.
- This exam is out of 40 points.
- This is a closed book examination.

I- True or False (4 Points)

1- It is not possible to combine functional paradigm with object-oriented paradigm.
False

2- Prolog is an imperative logic programming language
False

3- Lisp is side effect free
False

4- In Lisp `equal` and `eql` are not the same
True

II- Exercises on Prolog

1- Write down a Prolog program that reverses a list. (4 Points)

Examples
[1, 2, 3] → [3, 2, 1]
[4, [2, 3], 1] → [1, [2, 3], 4]
[7] → [7]

```
1- rev([], []).
```

```
2- rev([H|T], L) :- rev(T, RL), append(RL, [H], L).
```

2- Write down a Prolog program that from a list it produces the reverse list concatenated with the original list. (Hint: you can use the previous program) (4 Points)

Examples
[1, 2, 3] → [3, 2, 1, 1, 2, 3]
[1, [2, 4], 3] → [3, [2, 4], 1, 1, [2, 4], 3]

```
1- rev+(L1, L2) :- rev(L1, L3), append(L3, L1, L2).
```

3- Write down a Prolog program that reverses a list but also the internal lists. (4 Points)

Examples
[1, 2, 3] → [3, 2, 1]
[1, 2, [3, 8], 7] → [7, [8, 3], 2, 1]

```
1- rev1([], []).
2- rev1(X, X):- X \= [_|_].
3- rev1([H|T], L):-rev1(T, RL), rev1(H, H1), append(RL, [H1], L)
```

III- Exercises on Common Lisp

1- Give the output of the following Lisp functions: (3.5 Points)

a- `cdr '(10 20 30)`
(20 30)

b- `car '(10 (20) (30))`
10

c- `cdr '(10 (20) (30))`
((20) (30))

d- `cdr (cdr '(10 (20) (30)))`
((30))

e- `car (cdr (cdr '(10 (20) (30))))`
(30)

f- `cdr (cdr '(40 (50 (20) (30)) (60 (80) (40))))`
((60 (80) (40)))

g- `car (cdr (cdr '(40 (50 (20) (30)) (60 (80) (40)))))`
(60 (80) (40))

2- Let us consider the following program (8.5 Points)

```
 1-(defun add (number tree)
 2-  (if (null tree)
 3-      (list number)
 4-    (cond ((< number (car tree))
 5-           (list (car tree)
 6-                 (add number (car (cdr tree)))
 7-                 (car (cdr (cdr tree)))))
 8-          ((> number (car tree))
 9-           (list (car tree)
10-                 (car (cdr tree))
11-                 (add number (car (cdr (cdr tree))))))
12-          (t tree))))
```

a- What does `list` of line 3 perform?

It creates a list with the argument `number`. When the tree is empty, the argument number forms the tree.

b- What does `list` of line 5 perform?

It creates a list having three elements, 1) the root, 2) the left side, and 3) the right side.

c- Why we do not have `list` in line 6 before `car`?

Because `(car (cdr tree))` returns a list (a sub-list), which is the left side of the tree, so we don't need to convert this part to a list.

d- What does this program do? (Explain the details)

This program adds a number to a tree, which is a list having three components: 1) root, 2) left side, and 3) right side. The tree is represented as follows:
(Root Left_subtree Right_subtree). By adding an element, the function shecks that the elements of the Left_subtree are less than the Root and the Root is less than the elements of the Right_subtree. If the element we want to add already exists, the same tree is returned.

e- Illustrate with an example how this function can be executed in LispWorks (i.e. how it should be called). Give also the output of your call. In your example the second argument (tree) should have at least 5 elements.

Add 90 '(50 (40 (30) (45))(60 (55) (65)))

Output: (50 (40 (30) (45))(60 (55) NIL (90))))

We have: (car tree) = 50
90 > 50
So we execute:

```
 9-(list (car tree)
10-      (car (cdr tree))
11-      (add number (car (cdr (cdr tree))))))
```

(car tree) = 50
(car (cdr tree)) = (40 (30) (45))
(car (cdr (cdr tree))) = (60 (55) (65))

→ (50 (40 (30) (45)) (add 90 '(60 (55) (65))))
We have: (car tree) = 60
90 > 60
So we execute:

```
 9-(list (car tree)
10-      (car (cdr tree))
11-      (add number (car (cdr (cdr tree))))))
```

(car tree) = 60
(car (cdr tree)) = (55)
(car (cdr (cdr tree))) = (65)
→ (50 (40 (30) (45)) (60 (55) (add 90 '(65))))
We have: (car tree) = 65
90 > 65
So we execute:

```
 9-(list (car tree)
10-      (car (cdr tree))
11-      (add number (car (cdr (cdr tree))))))
```

(car tree) = 65
(car (cdr tree)) = Nil
(car (cdr (cdr tree))) = Nil

→ (50 (40 (30) (45))(60 (55) NIL (add 90 Nil))))
→ **(50 (40 (30) (45))(60 (55) NIL (90))))**

3- Write down a Lisp function that takes as argument a list and returns its reverse. (2 Points)

Examples
(1 2 3) → (3 2 1)
(4 (2 3) 1) → (1 (2 3) 4)
(7) → (7)

```
(defun reverse2 (lst)
  (cond ((null lst) '())
        (t (append (reverse2 (cdr lst)) (list (car lst))))))
```

4- Write down a Lisp function that reverses a list but also the internal lists. (5 Points)

Examples
(1 2 3) → (3 2 1)
(1 2 (3 8) 7) → (7 (8 3) 2 1)

```
(defun reverse3 (lst)
  (cond ((null lst) '())
        ((listp (car lst))
          (append (reverse3 (cdr lst)) (list (reverse3 (car lst)))))
        (t (append (reverse3 (cdr lst)) (list (car lst))))))
```

5- How can we access data in CLOS? (2 Points) (Not included in this year exam)

1- Using `slot-value`
2- Using `accessor`

6- Define a class `set` and a method `intersec (X Y Z)` such that X = Y ∩ Z. (3 Points)

```
(defclass set1 ()
  ((elmnts   :accessor set-elmnts
             :initarg :elmnts
             :initform '())))


   (defmethod intersec ((X set1) (Y set1) (Z set1))
     (setf (set-elmnts X)
             (setintersection1 (set-elmnts Y) (set-elmnts Z))))



(defun setintersection1 (lst1 lst2)
  (cond
    ((null lst1) '())
    ((null lst2) '())
    ((member (car lst1) lst2) (cons (car lst1)
          (setintersection1 (cdr lst1) lst2)))
    (t (setintersection1 (cdr lst1) lst2))))
```

Run :

```
(setq a (make-instance 'set1 :elmnts '(1 2 3)))
(setq b (make-instance 'set1 :elmnts '(1 2 4)))
(setq c (make-instance 'set1))
(intersec c a b)
→ (1 2)
```