UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in **INF2810**
Day of exam: **June 5, 2014**
Exam hours: **14:30 (4 hours)**
This examination paper consists of 4 pages.
Appendices: **None**
Permitted materials: **None**

Make sure that your copy of this examination paper is complete before answering.

Final Exam: INF2810, Spring 2014

Guidelines

- We recommend that you read through the full exam before you start (4 pages). In case you feel there might be missing information somewhere in the exam text, make your own assumptions and explain these briefly.
- Where we ask you to *write code* or *implement* something, we expect Scheme (or more specifically, R5RS as we have used throughout the semester). In case you get stuck on specifics of Scheme syntax or individual procedure names, however, it can be preferable to write pseudo-code, rather than writing nothing.
- Like in the lecture notes we will sometimes use "→" to indicate the value a given expression evaluates to.

1 List structures (16 points)

(a) Given the definitions below, what are the values of `z1` and `z2`? Draw box-and-pointer diagrams that show the underlying structures as well.

```
(define z1
  (let ((foo (list 'a 'b)))
    (cons foo foo)))

(define z2
  (cons (list 'a 'b) (list 'a 'b)))
```

(b) Again given the definitions above, explain the effect of the following call (you can also show this by drawing a box-and-pointer diagram if you prefer).

```
(set-car! z2 (cdr z2))
```

(c) Define a recursive procedure `nested-count` taking a symbol and a possibly nested list as arguments, returning the number of occurrences of the symbol in the list. Example call:

```
(nested-match 'b '((b) ((b a) b) a))
→ 3
```

(d) What type of process will your implementation of `nested-match` generate for the call example above?

2 let and lambda (7 points)

Rewrite the following expressions to an equivalent form that uses `lambda` instead of `let`. In addition, state the return value or effect that the expressions have when evaluated.

(a)
```
(let ((foo (list 1 2))
      (bar (* 2 2)))
  (cons bar foo))
```

(b)
```
(let ((foo (list 1 2)))
  (display foo)
  (newline)
  (let ((foo (cons 0 (cdr foo))))
    (display foo)))
```

3 Procedures (12 points)

(a) Write a procedure `compose` that takes two procedures as arguments – let's call them `p1` and `p2` – and returns a new procedure that applies `p1` to the result of applying `p2` to its argument. Both `p1` and `p2`, and the new procedure that is returned, expect a single argument. Example call:

```
(define (add1 x) (+ x 1))

(define (add100 x) (+ x 100))

((compose add1 add100) 5) → 106
```

(b) Based on `compose` you shall now write a procedure `repeat` that takes a procedure `p` and a positive integer `n` as arguments, and returns a new procedure that applies `p n` times. Example call:

```
((repeat add1 10) 20) → 30
```

(c) Define `eval-infix`, taking as argument a three-element list on the form *(arg1 operator arg2)*, and returning the value of applying the operator in the middle to the two operands `arg1` and `arg2`. Example calls:

```
(define exp1 (list 1 + 3))

(define exp2 (list 10 / 5))

(eval-infix exp1) → 4

(eval-infix exp2) → 2
```

4 Paradigms and idioms (25 points)

For this question you will write some different versions of a simple procedure `scale`. The arguments will be a number `x` and a list of numbers `seq`, and the return value a list where every element of `seq` has been multiplied by `x`. Example call:

```
(define foo (list 1 2 3 4))

(scale 3 foo) → (3 6 9 12)
```

(a) Write a purely functional version of `scale` based on tail recursion.

(b) Write a purely functional version of `scale` based on ordinary recursion.

(c) Write a purely functional version of `scale` based on higher-order sequence operations (it's fine to use built-in procedures here).

(d) Write a destructive version `scale!` that modifies its list argument.

(e) Write a stream version of you solution for (b) above: It should take a stream of numbers as argument and return a new stream of the scaled elements. You can here assume that the entire interface for working with streams that we have used in the course is available.

(f) Considering the example call on the sequence `foo` above, how many `cons` operations are spawned by calling your respective `scale` procedures from (b), (d) and (e)? For (e) you should assume that the sequence is a stream instead of a list, but with the same four elements.

5 Functional procedures (6 points)

Two built-in procedures in Scheme are `for-each` and `map` which we can think of as implemented like shown below (slightly simplified). Spend a sentence or two explaining the similarity and, more importantly, the difference between the two higher-order procedures as shown here. An important difference between functional and non-functional procedures is reflected in the difference between `for-each` and `map`; briefly explain what we are referring to here.

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))


(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
```

6 Environments (10 points)

We here turn to the environment model for evaluation. Draw an environment diagram showing all relevant frames and bindings after all the expressions in the following sequence have been evaluated.

```
(define items '(a b))

(define (keeper x)
  (set! items (cons x items))
  items)

(define (make-keeper items)
  (lambda (x)
    (set! items (cons x items))
    items))

(define k1 (make-keeper '(c d)))

(k1 'e)

(keeper 'f)
```

7 Encapsulation (17 points)

Write a procedure `make-accumulator` that returns a new procedure that encapsulates a local variable `sum` (initialized to 0) and lets us add a number (with the message `'add`) or subtract a number (with the message `'sub`). The procedure should also accept the message `'undo`, letting us undo a given number of previous calls and restore the sum to what is was before. In all cases the updated sum should be returned. Feel free to define additional helper procedures if you like. Example calls:

```
(define acc (make-accumulator))

(define acc2 (make-accumulator))

(acc 'add 5) → 5

(acc 'add 15) →  20

(acc 'add 80) →  100

(acc 'add 10) →  110

(acc 'sub 20) →  90

(acc 'undo 3) →  20

(acc2 'sub 5) →  -5
```

8 Evaluation strategies (7 points)

Towards the end of course we looked at how the metacircular Scheme evaluator could be modified to implement a form of so-called *normal-order evaluation* (*lazy evaluation*) as its standard evaluation strategy (for non-primitive procedures). As part of this we also chose to *memoize* the evaluation of expressions that denote the arguments for a procedure. Briefly explain the motivation for doing this. Also explain why it was not relevant to do this as long as the evaluator stuck to so-called *applicative-order evaluation* (*eager evaluation*) as its standard evaluation strategy.