UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in INF2810
Day of exam: Friday 5 June 2015
Exam hours: 14:30 (4 hours)
This examination paper consists of 4 pages (excluding this page).
Appendices: None
Permitted materials: None

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

Final Exam, INF2810, Spring 2015

Guidelines

- We recommend that you read through the full exam before you start (4 pages). Should you feel that you lack information for solving a specific problem, make your own assumptions and state them as part of your answer.
- Where we ask you to *write code* or *implement* something, we expect Scheme (or more specifically, R5RS as we have used throughout the semester). However, should you get stuck on specifics of Scheme syntax or certain procedure names, then writing pseudo-code with comments is preferable to writing nothing.
- Like in the lecture notes we will sometimes use "→" to indicate the value a given expression evaluates to.
- The maximum possible points indicated for each section sums to 100.

1 Pairs, lists and recursion (25 points)

(a) Carefully consider the sequence of expressions below. Draw a box-and-pointer diagram showing the structure that `foo` is bound to after the final expression is evaluated.

```
(define foo (list "bring" "towels" "!"))

foo → ("bring" "towels" "!")

(set-car! foo foo)

(set-cdr! (cdr foo) 42)
```

(b) We will here assume that `list?` is not a built-in primitive. Implement the predicate `list?` yourself. It should take a single argument and return `#t` if it's a list and `#f` otherwise. To earn the maximum points possible, avoid using `if` and `cond`. You do not need to consider the possibility of infinite cyclic structures as input.

(c) What do you think your `list?` predicate should evaluate to if applied to `foo` (i.e., after the entire sequence in 1a is evaluated)? Explain why in a single sentence.

(d) We will here work with nested lists, i.e., lists that may contain other (possibly nested) lists. Write a recursive procedure `deep-map` taking two arguments: a procedure `proc` and a (possibly nested) list `nested`. It should return a new nested list with `proc` applied to every atomic element of `nested`. Implement your procedure without using the built-in `map`. Example call:

```
(define nested '(((1) 2) 3 (4 (5 6))))

(deep-map (lambda (x) (* x 10))
          nested)

→ (((10) 20) 30 (40 (50 60)))
```

(e) Given the example call above, what type of process is spawned by your procedure `deep-map`? What can you say about the time complexity (growth in the number of steps).

2 Paradigms and idioms (25 points)

For this question you will write some different versions of a simple procedure `replace`. You should read through all the sub-questions before you start. The arguments will be two symbols `x` and `y` and a list of symbols `seq`. The return value should be a list where every occurrence of `x` has been replaced with `y`, while all other elements remain the same. Example call:

```
(define foo '(a b b a))

(replace 'a 'c foo) → (c b b c)
```

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

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

(c) Write a purely functional version of `replace` based on higher-order sequence operations (feel free to use built-in procedures).

(d) Write a destructive version of `replace` that modifies its list argument (and returns the modified list).

(e) Write a stream version of your solution for (b) above: Both the argument and the return value should be a stream of symbols. 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 `replace` 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.

3 Functional procedures (12 points)

(a) In just a couple of sentences, briefly explain what sets a purely functional procedure apart from a non-functional procedure.

(b) The built-in construction `begin` lets us evaluate a sequence of expressions. The value of the final expression in the sequence is also the value returned by the `begin` expression. This construction can be handy in combination with for instance `if`-expressions, which expects each argument to be one single expression, see the example call below.

Consider the following statement: As long as we stick to purely functional code we have no use for `begin`. Briefly explain why you believe this to be correct or not correct.

```
(if (some-test)
 (begin
   (do-something-1)
   (do-something-2))
 (do-something-3))
```

(c) In one sentence, explain the meaning of the term *higher-order procedures*.

4 Encapsulation and environments (16 points)

(a) We have seen that we can implement an abstract data type for pairs / cons-cells based on procedures. We will here assume that the interface is defined as follows (i.e., we will pretend that pairs are not a built-in data type).

```
(define (cons x y)
  (lambda (message)
    (cond ((eq? message 'car) x)
          ((eq? message 'cdr) y))))

(define (car p)
  (p 'car))

(define (cdr p)
  (p 'cdr))
```

Here are a few example calls, given the definitions above.

```
(define foo (cons 1 (cons 2 3)))

(car foo) → 1

(car (cdr foo)) → 2

(cdr (cdr foo)) → 3
```

Draw an environment diagram showing the value that `foo` is bound to, given the interaction above. (You do not have to show the bindings of other variables in the global environment, like `cons`, `car`, etc.).

(b) One thing missing from our interface so far is support for the destructive procedures `set-car!` and `set-cdr!` that let us change what values are stored in pairs. Add definitions for these to our interface. You will also need to extend the definition of `cons` above. Your implementation can use the built-in `set!` but not the two mutators that you are to define yourself. As an example, we want to be able to have the following kind of interaction:

```
(car foo) → 1

(set-car! foo 'a)

(car foo) → a

(set-cdr! (cdr foo) 'c)

(cdr (cdr foo)) → c
```

5 More encapsulation (10 points)

This question is about representing queues. A queue is here defined to have maximal length n. If a queue is n elements long when a new element is added, another element is automatically removed. The queues will follow the principle of first-in-first-out: New elements are added to the rear and if necessary the oldest element will be removed from the front of the list.

Define a procedure `make-queue` that takes one argument, an integer `n`, and returns a procedure object that via encapsulation lets us store up to n elements at a time in a list. Assume that `n` > 1. Example calls:

```
(define foo (make-queue 3))

(define bar (make-queue 2))

bar → #<procedure>

(bar 'a) → (a)

(bar 'b) → (a b)

(bar 'c) → (b c)

(bar 'd) → (c d)

(foo 1) → (1)

(foo 2) → (1 2)

(foo 3) → (1 2 3)

(foo 4) → (2 3 4)
```

We see that `make-queue` returns one-argument procedures that we here bind to `foo` and `bar`. Every call to `foo` and `bar` adds a new element and returns an updated list of elements. The queue should not grow longer than what we have specified and if necessary the oldest element is removed to make room for a new one.

The time complexity of storing and removing elements should be constant and independent of the size of n.

6 Streams and evaluation strategies (12 points)

(a) We will here assume that the full interface for working with streams (as it we know it from the course) is already defined and available to us. I addition we define a procedure `add-inf-streams` as follows:

```
(define (add-inf-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
               (add-inf-streams (stream-cdr s1) (stream-cdr s2))))
```

The recursive procedure `add-inf-streams` expects two infinite streams as arguments and returns a new stream with the sums of the pairwise elements from the two input streams. The procedure is well-defined as it stands (in the sense that it will work as intended if we attempt to call it). However, compared to most other recursive procedure definitions we have looked, an important element is left out in this definition. What is missing, and why is it still well-defined?

(b) Let us say we replace `cons-stream` with `cons` in `add-inf-streams` above, but otherwise let the definition remain unchanged. What will now be the result of calling `add-inf-streams`? Explain your answer.

(c) Carefully consider the expressions in the sequence below. What will be the result of evaluating the first expression? What will be the result of evaluating the second expression? Briefly explain why for both cases.

```
(define ones (cons-stream 1 ones))

(define integers (add-inf-streams ones integers))
```