UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in INF2810, Functional Programming
Day of exam: Friday 10 June 2016
Exam hours: 14.30
This examination paper consists of 5 pages (excl. front page).
Appendices: None
Permitted materials: None

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

Final Exam, INF2810, Spring 2016

Guidelines

- We recommend that you read through the full exam before you start (5 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 that a given expression evaluates to.
- The maximum available points indicated for each section sum to 100.

1 Pairs and lists (10 points)

(a) Carefully consider the sequence of expressions below. Draw a box-and-pointer diagram showing the structures that `foo` and `bar` are bound to after the final expression is evaluated. Show what the values will look like when printed at the REPL as well.

```
(define foo '(2 3 3 4))

(define bar (cons (car foo) (cdr foo)))

(set-car! foo 1)

(set-car! (cdr foo) 2)

(set! bar (cons 0 bar))
```

(b) Which Scheme procedures would have to replace `???` in the expression below, for it to evaluate to `#t`?

```
(equal? foo (??? 1 (??? bar))) → #t
```

2 Mystery (6 points)

In a sentence or two, explain the working of the following procedure. Try to employ terminology used in the class for your explanation. Additionally, give at least one concrete calling example and the corresponding return value, showing how the procedure can be used (i.e. you have to choose for yourself what the arguments are to be).

```
(define (x y z)
  (cond ((null? z) '())
        ((y (car z))
          (cons (car z)
                (x y (cdr z))))
        (else (x y (cdr z)))))
```

3 `take`, `drop` and `split-every` (27 points)

(a) Write a functional recursive procedure `take` that takes two arguments, a number `n` and a list `l`, and returns a new list with the `n` first elements of `l`. If `n` is larger than the length of `l`, the returned list will contain all the elements from `l`. Write *two* different versions; with and without tail-recursion. (Include a note indicating which version you take to be tail-recursive.) Example calls:

```
(define foo '(a b c d e))

(take 3 foo) → (a b c)

(take 0 foo) → ()

(take 7 foo) → (a b c d e)
```

(b) We will now write a non-functional variant of `take` where the goal is to *not* create new cons-cells (pairs). The procedure should work similarly to `take` in question *a* above in that it returns a list with the first `n` elements of a given list `l`, but this time it is allowed to destructively modify the list argument *l* to avoid generating new cons-cells. Note that, even though the procedure is granted the possibility to modify its argument, the idea is still that it is called for the sake of its return value. (Read question *c* below, before you start.) Example calls:

```
(define foo '(a b c d e))

(take 3 foo) → (a b c)

foo → (a b c)
```

(c) Explain what happens when your destructive version of `take` from question *b* is called with $n = 0$. What is the value of `foo` after the example call below? Explain your answer with just a couple of sentences.

```
(define foo '(a b c d e))

(take 0 foo) → ()

foo → ???
```

(d) Write a functional recursive procedure `drop` that takes two arguments, a number `n` and a list `l`, and returns what remains of the list after the `n` first elements. The procedure should not create a new list but rather return a suffix of the given list argument. If `n` is larger than the length of `l`, the empty list is returned. Example calls:

```
(define foo '(a b c d e))

(drop 1 foo) → (b c d e)

(eq? (cdr foo) (drop 1 foo)) → #t

(drop 3 foo) → (d e)

(drop 10 foo) → ()

(drop 0 foo) → (a b c d e)
```

(e) What type of process will be generated by a call to your procedure `drop` from question *d* above?

(f) Given the procedure `drop` from question *d* above, what is the value of `bar` after the calls shown below? Briefly explain your answer.

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

(define bar (drop 2 foo))

(set-cdr! bar foo)

bar → ???
```

(g) Based on what you have done so far in this section, write a functional procedure `split-every` that takes a number `n` and a list `l` and returns a list of sub-lists from `l` of length `n`. If the length of `l` is not divisible by `n`, the last sub-list will be shorter. This is illustrated in the example call:

```
(split-every 3 '(a b c d e f g h)) → ((a b c) (d e f) (g h))
```

4 Procedure-based object orientation (25 points)

(a) In this section, we will model local state and implement a type of objects that can be used at coffee bars: A card that has similarities to a bank account, where one can deposit money 'on' the card, to then use it to buy one or more cups of coffee. Each cup has a fixed price, which is stored on the card. Furthermore, the card provides a bonus programme, such that each n-th cup of coffee is free of charge; the specific value of n is also recorded on the card and initialized (once and for all) when the card is issued. As the card is used to order coffee, the return value indicates how many cups have been charged. In case the balance on the card is insufficient, the order is rejected; in this case, the return value will be `0`, and the balance and the count of previously paid cups of coffee, as recorded on the card, remain unchanged. Our card objects are initialized through a procedure `make-card` that takes three arguments: the initial value (or balance), the price for one cup, and the threshold n for the bonus programme. Following are some usage examples that further demonstrate the expected interface:

```
(define oe (make-card 100 25 5))

(define erik (make-card 100 20 7))

(card-order oe 1) → 1

(card-value oe) → 75

(card-count oe) → 1

(card-order oe 4) → 4

(card-count oe) → 0

(card-order erik 1) → 1

(card-deposit oe 50)

(card-order oe 1) → 1
```

```
(card-value oe) → 25
```

Some hints: It may be sensible to implement the abstract data-type using an interface based on *message passing*. Here, one needs to support a variable number of arguments, depending on specific messages. To realize the 'bonus programme', we recommend use of the procedures `quotient` and `remainder`, which implement integer arithmetic, e.g.

```
(quotient 7 3) → 2
(remainder 7 3) → 1
```

(b) In a few sentences, explain how we model local state 'on the card' in Scheme, and how each card gets to have their own, 'private' values for price, count, etc.

(c) In case your card implementation is internally based on an interface using *message passing*, which advantages do you see in 'hiding' the messages through an interface layer as demonstrated in our example above.

5 Recursion and streams (20 points)

A 'classical' algorithm to enumerate prime numbers is known as the *Sieve of Eratosthenes*, which takes as its point of departure a sorted, ascending list of integers starting from `2`. The first list element is always expected to be a prime number; at each step, successive numbers that are multiples of the first element are removed, making the immediately following number the next prime. This algorithm can be realized in pure functional style, using the higher-order procedure `filter` that we have discussed in more than one lecture. For example:

```
(define (divisible? numerator denominator)
  (zero? (remainder numerator denominator)))

(define (range i n)
  (if (> i n)
      '()
      (cons i (range (+ i 1) n))))

(range 2 7) → (2 3 4 5 6 7)

(define (eratosthenes n)
  (define (iter list)
    (if (null? list)
        '()
        (let* ((first (car list))
               (predicate (lambda (i) (not (divisible? i first)))))
          (cons first (iter (filter predicate (cdr list)))))))
  (iter (range 2 n)))

(eratosthenes 8) → (2 3 5 7)
```

(a) How often is `filter` invoked in the above calling example to `eratosthenes`, i.e. for $n = 8$?

(b) Write a procedure `stream-filter` that takes two arguments—a predicate and a stream—i.e. a stream-based variant of `filter`. You can assume the stream interface from the SICP book as available pre-defined, i.e. `cons-stream`, `stream-car`, `stream-cdr`, `stream-null?`, and `the-empty-stream`.

(c) When using streams, we need not impose an upper limit n for the enumeration of primes. Explain in a few sentences why this limitation is required in the list-based variant, and why it becomes unnecessary when we use streams.

(d) We have repeatedly noted a 'recipe' for converting a procedure from operating over lists to working with streams. Implement `stream-range` og `stream-eratosthenes`. In these stream-based versions, the code can be substantially simplified because the upper bound n shall be removed (an intervall can of course be infinitely open-ended).

(e) `(stream-cdr (stream-cdr (stream-eratosthenes)))`

6 Theory (12 points)

Briefly discuss what you take to be central characteristics of functional code. (Do not spend more than half a page.)