Solutions

# CMPT 383 Midterm Sample
# Summer 2019

| | |
|---|---|
| **Last** name *exactly as it appears on student card* | |
| **First** name *exactly as it appears on student card* | |
| **SFU Student #** | |
| **SFU email** *ends with* `sfu.ca` | |

This is a **closed book exam**: notes, books, computers, calculators, electronic devices, etc. are **not permitted**. Do not speak to any other students during their exam or look at their work. If you have a question, please remain seated and raise your hand and a proctor will come to you.

| | ***Out of*** | ***Your Mark*** |
|---|---|---|
| *Deep Functions* | 10 | |
| *Folding* | 10 | |
| *Short Answer: Scheme* | 6 | |
| *Currying and Continuations* | 10 | |
| *Haskell* | 10 | |
| Total | 46 | |

## A Counter

(10 marks) Write a "deep" function called `(acount x)` that works as follows:

- If `x` is the symbol `'a`, then `1` is returned.
- If `x` is a non-list other than `'a`, then `0` is returned.
- If `x` is a list, it returns the number of occurrences of the symbol `'a` (even ones deeply nested within sub-lists) in x.

For example:

```
> (acount 'a)
1

> (acount'm)
0

> (acount '(a big (dog is a handful)))
2

> (acount'((a) (1 (((a)) dog) a)))
3
```

Use only basic Scheme functions that are part of standard Scheme and were discussed in the lectures and notes (no loops!).

Make sure to use good Scheme programming style and perfect indentation: make your code easy to read.

```
(define (sum lst)
    (cond ((null? lst)
              0)
          (else
              (+ (car lst) (sum (cdr lst))))
    )
)

(define (acount x)
    (cond ((equal? x 'a)
              1)
          ((not (list? x))
              0)
          (else
              (sum (map acount x))
          )
    )
)
```

Other solutions are possible!

## Let and Functions

a) (1 mark) Write the most general form of a Scheme `let` expression.

```
(let ((v1 val1) (v2 val2) … (vn valn)) body)
```

b) (2 marks) Rewrite the following expression as an equivalent one that *doesn't* use `let`, and *doesn't* use any functions or special forms except for `lambda`:

```
(let ((x 2)
      (y 3))
   (* x y))
```

```
((lambda (x y) (* x y)) 2 3)
```

c) (2 marks) Rewrite the following expression as an equivalent one that *doesn't* use `let*`, and *doesn't* use any functions or special forms except for `let`:

```
(let* ((x 2)
       (y x))
   (* x y))
```

```
(let ((x 2))
   (let ((y 2))
       (* x y)))
```

d) (5 marks) Consider the following function:

```
(define my-if
   (lambda (test true-body false-body)
      (cond (test true-body)
            (else false-body))))
```

Can this be used in place of Scheme's built-in `if`? Justify your answer.

No, this cannot be used as a replacement for Scheme's built-in `if` form. Scheme's built-in `if` is a macro (or special form), and it only evaluates `true-body` when `test` is true, and it only evaluates `false-body` when `test` is false. The built-in `if` never evaluates both `true-body` and `false-body`.

In contrast, `my-if` is a function (not a macro) and so it evaluates both `true-body` and `false-body` before evaluating the `cond` expression. This is a big difference!

For example, suppose you are using Scheme to control a robot that can turn left or right. The expression `(my-if (wall-on 'left) (turn_right) (turn_left))` would evaluate both `(turn_right)` and `(turn_left)`, i.e. the robot would turn both left and right. In contrast,
`(if (wall-on 'left) (turn_right) (turn_left))` would correctly just make the robot turn left.

## Short Answer: Scheme

| | |
|---|---|
| a) (1 mark) *True* or *False*: if you evaluate the following expression in the Scheme interpreter, you will get an error:<br>`(list list list)` | *False* |
| b) (1 mark) *True* or *False*: if you evaluate the following expression in the Scheme interpreter, you will get an error:<br>`(cons cons cons)` | *False* (it returns a *pair* of functions) |
| b) (1 mark) *True* or *False*: if you evaluate the following expression in the Scheme interpreter, you will get an error:<br>`(cdr cdr cdr)` | *True* |
| c) (3 marks) Is it possible to create a Scheme expression `f` such that `((f))` evaluates to 5? If so, define such an `f`. If it is not possible, explain why. | `(define f`<br>`  (lambda ()`<br>`      (lambda () 5)))` |

## Currying and Continuations

(10 marks) Implement the following function using continuation passing style (CPS) and show how to call the CPS version to calculate `(hyp 2 3)`.

```
(define hyp
    (lambda (a b)
        (sqrt (+ (* a a) (* b b)))
    )
)
```

```
(define (id x) x)
(define (+c a b k) (k (+ a b)))
(define (*c a b k) (k (* a b)))
(define (sqrt-c a k) (k (sqrt a)))
```

```
(define (hyp-c a b k)
    (*c a a (lambda (as)
                (*c b b (lambda (bs)
                            (+c as bs (lambda (total)
                                          (sqrt-c total k)))))))
)
```

Sample call of `hyp-c`: `(hyp-c 2 3 id)`

## Haskell

(10 marks) Write a function (including its most general signature) called `swapEnds` that takes a list as input and returns a new list that is the same as the input one except the first element and the last element have been swapped. If the list as fewer than 2 elements, then its returned unchanged.

For example:

```
> swapEnds "yogurt"
"togury"

> swapEnds [1,2,3,4]
[4,2,3,1]

> swapEnds ["up","down"]
["down","up"]

> swapEnds "M"
"M"
```

To get full marks, *don't* use recursion in your solution. **Hint**: The standard Haskell function `last lst` returns the last element of the list `lst`.

```
swapEnds :: [a] -> [a]
swapEnds lst
        | length lst < 2 = lst
        | otherwise      = [end] ++ middle ++ [begin]
                           where begin  = head lst
                                 end    = last lst
                                 middle = take (n-2) (tail lst)
                                 n      = length lst
```

other solutions are possible