

Assignment 1B

Due Friday February 13 at 11:50am, just before tutorial.

No late assignments will be accepted.

The questions below are designed to make you think hard about Scheme functions. Some of them are considerably harder than those in Assignment 1A, so be *sure to start the assignment early*. As before, your Scheme functions should be well commented, and should be written in good functional programming style. In particular, do *not* use any functions or constructs that change the values of variables or have other side effects or that use a sequential processing of commands. For example, you may not use `do`, `begin`, or any function ending in `!`, such as `set!`, `set-car!`, `vector-set!`, etc. Feel free to use helper functions wherever appropriate. You should hand in three files: a source listing of all your Scheme functions, a sample terminal session with the Scheme interpreter, and a text file containing solutions to all “pencil and paper” problems, including proofs. The terminal session should be short, and should demonstrate that your functions work correctly. The three files should be submitted electronically as described on the course web page. In general, simple solutions are preferred and will receive the most marks. To keep things simple, you may assume that the input to your functions is correct, so that no error checking is required.

The marker has a limited amount of time for each assignment, so it is your responsibility to provide documentation and testing that allows him to *quickly* evaluate your work. As with all work in this course, 20% of the grade is for quality of presentation.

1. (5 points total) Consider the following Scheme function:

```
(define (f N) (if (zero? N) 0 (+ N (f (- N 1)))))
```

Prove that this function has the following properties:

- (i) $(f\ N) = 2N + (f\ N-2) - 1$ for all $n \geq 2$ (2 points)
- (ii) $(f\ N) = N(N+1)/2$ for all $n \geq 0$ (3 points)

Justify each step of your proof.

2. (5 points) Consider the following Scheme functions:

```
(define (length x)
  (if (null? x) 0 (+ 1 (length (cdr x)))))

(define (append x y)
  (if (null? x) y (cons (car x) (append (cdr x) y))))

(define (reverse x)
  (if (null? x) '() (append (reverse (cdr x)) (list (car x)))))
```

Prove that these functions have the following property, for every list X :

$$(\text{length } (\text{reverse } X)) = (\text{length } X)$$

Justify each step of your proof. You may use properties of `append` and `length` proved in class or in the on-line notes.

3. (35 points total) In an s-expression, we define the *depth* of a symbol or a number to be the number of brackets that enclose it. For example, in the expression `(a (b (c d) e) f)`, the symbols `c` and `d` have depth 3; the symbols `b` and `e` have depth 2; and the symbols `a` and `f` have depth 1.

Actually, since the same symbol or number may appear more than once, it may have more than one depth. For instance, in the expression `(a (a (a a a)))`, the symbol `a` appears five times at three different depths, 1, 2 and 3. To account for this, we define `depths(s, e)` to be the set of depths of symbol (or number) `s` within expression `e`. For example,

$$\text{depths}(a, (a b c)) = \{1\}$$

$$\text{depths}(a, (a (b (a a)))) = \{1, 3\}$$

$$\text{depths}(a, (b c d)) = \{ \}$$

$$\text{depths}(a, a) = \{0\}$$

With this in mind, answer the following questions. Recall that simple solutions are preferred and will receive the most grades.

- (a) (5 points) Give a formal, recursive definition for `depths(s, e)`. (This is a pencil-and-paper exercise, not a programming exercise.)
- (b) (10 points) Define a Scheme function `(depths S E)` that computes the set of depths of symbol (or number) `S` within s-expression `E`. This set should be represented as a list with all duplicates removed. For example,

(depths 'a '(a b c)) => (1)

(depths 'a '(a (b (a a)))) => (1 3)

- (c) (10 points) Define a Scheme function (`remDepth D E`) that removes all symbols and numbers of depth `D` in s-expression `E`, for $D > 0$. For example,

(remDepth 1 '(1 (2 (3 4) 5 (6 7) 8) 9)) => ((2 (3 4) 5 (6 7) 8))

(remDepth 2 '(1 (2 (3 4) 5 (6 7) 8) 9)) => (1 ((3 4) (6 7)) 9)

(remDepth 3 '(1 (2 (3 4) 5 (6 7) 8) 9)) => (1 (2 () 5 () 8) 9)

(remDepth 4 '(1 (2 (3 4) 5 (6 7) 8) 9)) => (1 (2 (3 4) 5 (6 7) 8) 9)

(remDepth 1 '(a (a (a a) a) a)) => ((a (a a) a))

- (d) (10 points) Define a Scheme function (`remMax E`) that removes all symbols and numbers of maximum depth in `E`. For example,

(remMax '(a (b))) => (a ())

(remMax '((a) b)) => (() b)

(remMax '(a (b (c d)))) => (a (b ()))

(remMax '(a (((b c)))))) => (a (((())))

(remMax '(a (((b)))))) => (a (((())))

(remMax '(a (((()))) => (((())))

4. (30 points) A road map consists of a set of cities, and a set of roads connecting them. A road can be thought of as an (unordered) pair of cities. For example, here is a simple road map of Southern Ontario:

Cities = {Toronto, Ottawa, Kingston, Montreal}

Roads = {(Toronto, Montreal), (Toronto, Kingston),
(Kingston, Montreal), (Kingston, Ottawa),
(Ottawa, Montreal)}

Each road in a map is two-way, and it represents a direct connection between cities; i.e., it takes you from one city to another without passing through any other cities.

You are to write a Scheme function that helps a tourist to plan a vacation in which he visits each city exactly once. Specifically, given a road map `M`, a *path* is a list of cities (C_1, C_2, \dots, C_n) such that there is a road from `C1` to `C2`, another road from `C2`

to C3, etc. Your job is to define a Scheme function (`path M C`) that returns a path starting at `C` that mentions each city *exactly once*.¹ For example, if `M` is the road map of Southern Ontario given above, then (`path M 'Toronto`) might evaluate to the list `(Toronto, Kingston, Montreal, Ottawa)`. If such a path does not exist, then return `#f` or the empty list, `()`.

IMPORTANT:

- You should test your function on road maps having at least ten cities and fifteen roads.
 - Your documentation should provide a high-level description of the algorithm and data structures used.
5. (15 points) Define a function `compose` that takes a list of unary functions as input, and returns their composition as output. For example, (`compose (list car cdr)`) should return `(lambda (X) (car (cdr X)))`. Likewise, (`compose (list list car list cdr)`) should return `(lambda (X) (list (car (list (cdr X)))))`. It should be possible to apply the composed functions to an argument. For example,

```
(define f (compose (list cdr cdr cdr)))
(f '(a b c d e f)) => (d e f)

(define g (compose (list list list car cdr cdr)))
(g '(a b c d e f)) => ((c))

(define h (compose (list car cdr car cdr)))
(h '(1 (5 6 7) (7 8) 4)) => 6

(define k (compose (list square
                    (lambda (X) (+ 1 X))
                    square)))
(k 3) => (square (+ 1 (square 3))) => 100
```

In addition,

- (a) `compose` should be recursive, and it should *not* use any helping functions.
- (b) It should be possible to give `compose` the null list. i.e., the expression (`compose '()`) should return something reasonable.
- (c) `compose` should print the string `calling compose` each time it is called. Thus, the expression (`compose (list cdr cdr cdr)`) should print the following:

```
calling compose
calling compose
calling compose
calling compose
```

¹Since each city is visited exactly once, if road map `M` has `n` cities, then the list returned by (`path M C`) must have length `n`.

since, in addition to the top-level call, `compose` calls itself recursively three times.

- (d) The string `calling compose` should *never* be printed when a composed function is applied to an argument. For example, consider the following sequence of commands to the Scheme interpreter:

```
(define f (compose (list cdr cdr cdr)))
```

```
(f '(a b c d e f))
```

```
(f '(1 2 3 4))
```

```
(f '(1 a 2 b 3 c 4 d))
```

The first line should cause the string `calling compose` to be printed four times. However, the remaining three lines should *not* cause any strings to be printed. One way to achieve this is through an appropriate use of `let` expressions.

NOTE: In Scheme, the expression `(display s)` will print the value of `s`, and the expression `(newline)` will print a new line. Printing output is non-functional and requires sequential processing. There are several ways to force a Scheme function to execute statements sequentially. Perhaps the easiest is to define a function as follows:

```
(define (f X) e1 e2 ... en)
```

This will cause function `f` to execute the expressions `e1`, `e2`, ... `en` sequentially, in order. The value of `en` is then returned as the value of `(f X)`. The following definition does exactly the same thing:

```
(define f (lambda (X) e1 e2 ... en))
```

90 points total.

Cover sheet for Assignment 1B

Complete this page and attach it to the front of your assignment.

Name: _____
(Underline your last name)

Student number: _____

I declare that this assignment is solely my own work, and is in accordance with the University of Toronto Code of Behavior on Academic Matters.

Signature: _____