University of Toronto Mississauga
CSC 324 - Principles of Programming Languages, Fall 2016

Assignment 1

Due Tuesday October 11 at 11pm.
No late assignments will be accepted.

What to do

The questions below require you to write Scheme functions. The first few questions provide warm-up problems to get you use to the Scheme syntax, the programming environment and thinking like a functional programmer. In all questions, 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. To help you with this new way of thinking, I am disallowing the use of `do`, `begin`, or any function ending in `!`, such as `set!`, `set-car!`, `vector-set!`, etc. You must use recursion in *all* your solutions and use helper functions wherever appropriate. You may also find the following built-in Scheme functions useful: `list`, `append`, `reverse`, `length`, and `cadr`, `cddr`, `cadar`, `caddr`, etc. Other than these, *use only the functions mentioned in class*, unless specified otherwise. In particular, do not import any Scheme/Racket modules. Also, do not use any built-in function that would require recursion if you defined it yourself, unless specified otherwise. You may, of course, use any function you like if you define ityourself (in terms of allowed functions). The point here is that you should not scour the user manual or the web for functions that will solve most of a problem for you. Instead, your solution should use only basic functions, like `car`, `cdr`, `cons`, null?, `equal?`, `cond`, `if`, `and`, `or`, `lambda`, `+`, `-`, `>`, etc.

You should hand in four files: the source code of all your Scheme functions, a sample terminal session with the Scheme interpreter, the answers to the non-programming questions, and a scanned signed copy of the cover sheet at the end of the assignment. The source code should be well commented, and the terminal session should be short and should demonstrate that your functions work correctly. The files should be submitted electronically as described on the course web page. Be sure that we can run your Scheme code on the UTM computers. The terminal session should demonstrate that your functions work on all of the examples given in this assignment and on whatever other examples you deem necessary to demonstrate that your functions work correctly. (You will be graded on your choice of extra examples.)

Note: 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. 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 no error checking is required. As with all work in this course, 20% of the grade is for quality of presentation.

No more questions will be added

1. (5 points) Define a Scheme function `(sumAbs L)` that returns the sum of the absolute values of the numbers at the top level of list `L`. For example,

```
(sumAbs '(1 -5 -2 3))  =>  1+5+2+3 = 11

(sumAbs '(1 -5 a -2 b 3))  =>  1+5+2+3 = 11

(sumAbs '(1 (2 3) (a b) 5))  =>  1+5 = 6
```

   You may use the built-in Scheme function `abs`.

2. (7 points) Define a Scheme function `(countEven NL)` that returns the number of even numbers in nested list `NL`. The even numbers may occur at any depth and may be repeated. For example,

```
(countEven '(2 a 3 b c 4))  =>  2

(countEven '(2 (a (3 (b (c 4))))))  =>  2

(countEven '(2 (4 (2) 4) 2))  =>  5

(countEven '((((4)))))  =>  1

(countEven 4)  =>  1

(countEven 'a)  =>  0

(countEven '())  =>  0
```

   You may use the built-in Scheme function `even?`.

3. (7 points) Define a Scheme function `(getSymbols L)` that returns a list of all the symbols at the top level of list `L` in order. For example,

```
(getSymbols '(1 a 2 3 b a))  =>  (a b a)

(getSymbols '(1 2 3))  =>  ()

(getSymbols '(a (b c) (1 2) 3 d))  =>  (a d)
```

4. (7 points) Define a Scheme function `(prefix L A)` that returns all the elements of list `L` that precede the first occurrence of `A` at the top level of the list. If `A` does not appear at the top level of the list, then return the entire list. For example,

```
(prefix '(1 2 3 4 5 6) 4)  =>  (1 2 3)

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

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

(prefix '(a b c d) 'e)  =>  (a b c d)

(prefix '(a (b c) d b e) 'b)  =>  (a (b c) d)
```

5. (10 points) Define a Scheme function `(transform NL)` that replaces all negative numbers in `NL` by -1, and replaces all positive numbers by 1, and leaves all other elements unchanged. Here, `NL` is a nested list. For example,

```
(transform '(-4 -2 0 2 4))  =>  (-1 -1 0 1 1)

(transform '(a (-3 (0 () 4) b) -8))  =>  (a (-1 (0 () 1) b) -1)

(transform -8)  =>  -1

(transform '((((())))))  =>  ((((()))))
```

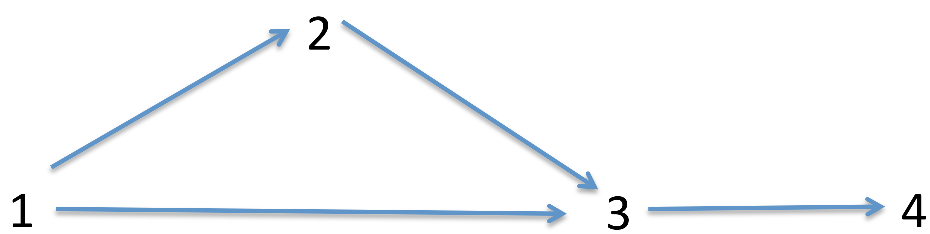
6. (7 points) Define a Scheme function `(map2 F L1 L2)` where `L1` and `L2` are lists of equal length. If `L1 => (a1 a2 ..  aN)` and `L2 => (b1 b2 ...  bN)`, then `map2` returns a list of length `N` who's ith element is `(F ai bi)`. For example,

```
(map2 + '(1 2 3 4) '(5 6 7 8))  =>  (1+5 2+6 3+7 4+8) = (6 8 10 12)

(map2 cons '(a b c) '((1 2) () (d e f)))  =>  ((a 1 2) (b) (c d e f))

(map2 (lambda (X Y) (+ 1 (* X Y))) '(1 2 3 4) '(5 6 7 8))
  =>  (1*5+1 2*6+1 3*7+1 4*8+1) = (6 13 22 33)
```

The next three questions require you to define functions on a directed graph. You should represent a graph as a list of edges, where each edge is a list of two nodes and each node is a number. For example, the list `(3 7)` represents an edge pointing from node 3 to node 7. The graph below is thus represented as the list `((1 2) (2 3) (3 4) (1 3))`. Note that each edge appears exactly once in the list and the order of the edges in the list is unimportant. Likewise, in each of the questions below, the order of nodes in a list does not matter, since the list represents a set.



You may find it useful to define functions that implement set operations. In this case, we implement sets as lists with no duplicate elements. For example, the list `(a b c)` would represent a set, whereas the list `(a b a)` would not. You could then define `(union S1 S2)` to return the union of sets `S1` and `S2`, or you could define `(intersect S1 S2)` to return their intersection. Likewise, you might want to define `(setDiff S1 S2)` to return the difference of sets `S1` and `S2`. (The set difference is all the elements in `S1` that are not in `S2`.)

Your solutions should be able to handle very large graphs. In particular, your functions should not run in exponential time. To test this, I have provided Scheme code for generating large graphs, which is available on the course web site. It generates a graph called `G4001` which has 4001 nodes. You should test your functions in Questions 8 and 9 on this graph using node `N = 4000` and depths `D = 20, 21, 22, 200, 201, 202`. Please include approximate execution times for these values of `D` when submitting your solutions. If your functions take more than 10 minutes to terminate and return the correct answer when `D = 200`, you will receive at most half points for the problem.

7. (7 points) If a graph contains an edge from node M to node N, then we say that N is a child of M. Define a Scheme function `(children N G)` that returns the children of node `N` in graph `G`. For example, if `G1 => ((1 2) (1 3) (2 4) (3 4))`, then

```
(children 1 G1)  =>  (2 3)
(children 2 G1)  =>  (4)
(children 3 G1)  =>  (4)
(children 4 G1)  =>  ()
```

8. (15 points) If a graph contains a sequence of m edges $(N_0 N_1)$, $(N_1 N_2)$ ... $(N_{m-1} N_m)$, then we say that the graph has a directed path of length m from node N_0 to node

N_m. Note that there may be many directed paths, of different lengths, from one node to another. If the shortest directed path from node M to node N has length D, then we say that N is distance D from M (or that N is a descendant of M of depth D). Note that because the paths are directed, the distance from M to N may not be the same as the distance from N to M.

Define a Scheme function `(descendantsAll N D G)` that returns a list of all the nodes in graph `G` that are distance `D` or less from node `N`. The list should not contain any duplicate nodes. For example,

```
(descendantsAll 1 0 G1)  =>  (1)
(descendantsAll 1 1 G1)  =>  (1 2 3)
(descendantsAll 1 2 G1)  =>  (1 2 3 4)
(descendantsAll 1 3 G1)  =>  (1 2 3 4)
(descendantsAll 1 7 G1)  =>  (1 2 3 4)
(descendantsAll 2 0 G1)  =>  (2)
(descendantsAll 2 1 G1)  =>  (2 4)
(descendantsAll 3 0 G1)  =>  (3)
(descendantsAll 3 1 G1)  =>  (3 4)
(descendantsAll 4 0 G1)  =>  (4)
(descendantsAll 4 1 G1)  =>  (4)
```

Likewise, if `G2 => ((1 2) (2 3) (3 4) (1 3))`, then

```
(descendantsAll 1 1 G2)  =>  (1 2 3)
(descendantsAll 1 2 G2)  =>  (1 2 3 4)
```

Hint: define a helper function that carries out breadth-first search in `G` starting at node `N`. (See "breadth-first search" in Wikipedia.)

For full marks, your definition of `descendantsAll` should be simpler than your definition of `descendants2` in Question 9(b). In Particular, you should not use `descendants2` to define `descendantsAll`.

9. (20 points total) Define a Scheme function `(descendants N D G)` that returns a list of all the nodes in graph `G` that are *exactly* distance `D` from node `N`. The list should not contain any duplicate nodes. For example,

```
(descendants 1 0 G1)  =>  (1)
(descendants 1 1 G1)  =>  (2 3)
(descendants 1 2 G1)  =>  (4)
(descendants 1 3 G1)  =>  ()
(descendants 1 7 G1)  =>  ()
(descendants 2 0 G1)  =>  (2)
(descendants 2 1 G1)  =>  (4)
(descendants 3 1 G1)  =>  (4)
(descendants 3 2 G1)  =>  ()
```

Likewise,

```
(descendants 1 1 G2)  =>  (2 3)
(descendants 1 2 G2)  =>  (4)
(descendants 1 3 G2)  =>  ()
```

Likewise, if `G3 => ((1 2) (2 3) (3 4) (4 5) (1 4))`, then

```
(descendants 1 1 G3)  =>  (2 4)
(descendants 1 2 G3)  =>  (3 5)
(descendants 1 3 G3)  =>  ()
```

Finally, if `G4 => ((0 1) (1 2) (2 3) (3 4) (4 5) (5 6) (6 7) (7 8) (1 3) (1 4) (1 5) (1 6))`, then

```
(descendants 0 0 G4)  =>  (0)
(descendants 0 1 G4)  =>  (1)
(descendants 0 2 G4)  =>  (2 3 4 5 6)
(descendants 0 3 G4)  =>  (7)
(descendants 0 4 G4)  =>  (8)
(descendants 0 5 G4)  =>  ()
```

You should define `descendants` in two different ways, using two different helper functions, as described below. In each case, `descendants` is a simple, non-recursive function and the helper function is recursive and does most of the work. The two versions should be called `descendants1` and `descendants2`.

(a) (5 points) Define `(descendants1 N D G)` using `descendantsAll` as a helper function.

(b) (15 points) Define `(descendants2 N D G)` so that it carries out a breadth-first search of graph `G` starting at node `N`. (See "breadth-first search" in Wikipedia.) You should do this by defining a helper function `(descendants2Help L1 L2 D G)`, where `L1` and `L2` are lists of nodes in `G`. At each recursive call to `descendants2Help`, `L2` contains all the nodes we have visited so far, and `L1` contains all the nodes we are now visiting for the first time. More specifically, at the n^{th} recursive call, `L2` contains all the nodes at distance less than n, and `L1` contains all the nodes at distance n. `descendants2Help` should be linear recursive, that is, each call to `descendants2Help` should give rise to at most one recursive call to itself. Do not use `descendantsAll` in your definition.

If the function call `(descendants2 4000 2000 G4001)` terminates with the correct answer in less than 10 minutes, you will receive 5 bonus points. To recieve the bonus points, we must be able to run your code.

10. (20 points total) Consider the following two Scheme functions:

```
(define (attach X L)
  (if (null? L)
      (cons X L)
      (cons (car L) (attach X (cdr L)))))

(define (member? X L)
  (cond ((null? L) #f)
        ((equal? X (car L)) #t)
        (else (member? X (cdr L)))))
```

(a) (5 points) Write down the basic properties of these two functions, i.e., properties that are immediately evident from inspection of the code.

Prove that these functions have the following properties, for all lists `L` and any expression `A`:

(b) (5 points) `(member? X (attach X (cons A '()))) = #t`

(c) (10 points) `(member? X (attach X L)) = #t`

Justify *every* step of your proofs. Do not use (c) to prove (b).

No more questions will be added

University of Toronto Mississauga
CSC 324 - Principles of Programming Languages

Cover sheet for Assignment 1

Complete this page and submit it with 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 Behaviour on Academic Matters.

Signature: ______________________________