COMP 3007 - Assignment #4

Scheme objects, and environments.

Due: ~~Friday March 20th @ 11:55pm~~ Sunday March 22nd @ 11:55pm

Question 1

[5 marks] Given the following code:

```
01| (define (run n)
02|   (define (iter a b c)
03|     (cond ((= c 0) b)
04|           ((> a b) (iter (- a b) (+ b a) (- c 1)))
05|           (else (iter (* a 2) (+ b 1) (- c 1)))))
06|   (if (< n 4)
07|       n
08|       (iter n 0 n)))
09|
10| (run 5)
```

Draw a contour diagram at the start of line 03 during the *second* evaluation of `iter` (ie. before the call to (iter 10 6 3)).

Question 2

[10 marks] Given the following code:

```
01| (define (f L)
02|   (cond ((null? L) '())
03|         ((< (car L) 0)(cons (* (g (cdr L)) (car L)) (f (cdr L))))
04|         (else (cons (car L)(f (cdr L))))))
05|
06| (define (g L)
07|   (cond ((null? L) 0)
08|         (else (+ (car L)(- (g (cdr L)))))))
09|
10| (f '(3 -7 4 2))
```

a. [5 marks] Draw a contour diagram during the evaluation of `g` at the beginning of line 8 (before g recurses on itself) using *lexical* (static) scoping rules.
b. [5 marks] Draw a contour diagram during the evaluation of `g` at the beginning of line 8 (before g recurses on itself) using *dynamic* scoping rules.

Question 3

[10 marks] Given the following code:

```
01| (define (alpha a)
02|   (define z 3)
03|   (define (beta b)
04|      (define a (+ z b))
05|      (+ a (gamma b)))
06|   (define (gamma c)
07|      (set! a (* z c))
08|      a)
09|   beta)
10|
11| (define omega (alpha 1))
12| (omega 2)
```

a. [/6 marks] Draw a contour diagram at the start of line 5 (after calling omega on line 12, before calling gamma on line 5). Your drawing should assume lexical scope.
b. [/1 mark] What is the output of this code? (Using lexical scoping)
c. [/3 marks] Would this code work using dynamic scope as taught in lecture? If yes, provide the output. If no, explain why not.

Question 4

[32 marks] Define a procedure called `make-graph` that implements a Graph ADT using object oriented design in Scheme. The `make-graph` procedure should return an *object* that supports the following operations via a message-passing style:

- `(add-node x)` - adds a given node (symbol/value) to the graph if it does not already exist.
- `(add-edge x y)` - adds a directed edge from node x to node y in the graph if both nodes exist and the edge does not.
- `(remove-node x)` - removes the given node from the graph if it exists. All links to the given node should be removed as well.
- `(remove-edge x y)` - removes the given edge from the graph if it exists.
- `(display)` - displays the contents of the graph to the user in the format: "node: edge edge edge", one node per line

With the exception of the display method, all other methods should return #t if successful and #f otherwise.

Your graph should use list(s) (and/or pairs) as the backing data structure, storing the graph in an adjacency list format.
For example: ((a . (b c)) (b . (c a)) (c . (a b c)) ...)
Note: order is not important.

You are encouraged to create any private helper functions or abstractions you may require to simplify the construction of the above methods.

Here is an example interaction with the completed solution:

```
(define G (make-graph))
((G 'add-node) 'a)        ;=> #t
((G 'add-node) 'b)        ;=> #t
((G 'add-node) 'c)        ;=> #t
((G 'add-node) 'a)        ;=> #f

((G 'add-edge) 'a 'b)     ;=> #t
((G 'add-edge) 'a 'c)     ;=> #t
((G 'add-edge) 'b 'b)     ;=> #t
((G 'add-edge) 'b 'c)     ;=> #t
((G 'add-edge) 'c 'd)     ;=> #f
((G 'display))            ;=> a: b c
                              b: b c
                              c:
((G 'remove-edge) 'a 'c) ;=> #t
((G 'remove-node) 'c)     ;=> #t
((G 'display))            ;=> a: b
                              b: b
```

Documentation & Testing

Documentation

- Ensure that your name and student number are in comments at the top of all files.
- Document the purpose of each function including its expected inputs (parameters) and output (return).
- Ensure that your code is well-formatted and easily readable; a happy TA is a generous TA.

Testing

- You are required to include testing runs of every function in your submission.
- The specific tests required depend on the question at hand, but should cover all valid inputs and all possible branches of your code.
- The example runs provided in the guidelines above may not be sufficient.
- Unless otherwise specified, you may assume inputs supplied are of the correct type.
- Fabricated test outputs will result in 0 marks for a question.
- For best practices: Comment your testing as to what you are testing and why, giving expected output as well as observed output and explanations for any differences.

[10 marks]

An example submission including documentation and testing can be found here: primes.scm

Submission

- **Submit contour diagrams as .pdf files.** Any other format will result in zero (0) marks for your submission. I recommend MS power point or open office impress for creating simple shapes with text, though you are free to use any drawing program you like. Again, be sure to save/print as .pdf.
- You may hand-draw and photograph your submissions. However, ensure your submissions are submitted as .pdf files, and that your work is legible. The TAs *will not* endeavour to interpret any hard to read text or shapes. Illegible submissions will result in loss of marks at the discretion of the grader.
- Any program files that are not runnable (in DrRacket using R5RS) will result in a mark of 0 for that question.
- You are responsible for submitting all files and ENSURING that the submission is completed successfully.
- Submit your assignment using **cuLearn**.
- Combine all files into a single zip file for the whole assignment.
- Marks will be deducted for late submissions.
- If you are having issues with submission, contact me **before** the due date, afterwards late deductions will apply.
- Please see the **course outline** for all submission guidelines.