Programming III COMP2209

Coursework 1

Individual Coursework *(cf. collaboration policy)*

- *Aims of this coursework:*
  1. *Programming with a Scheme environment.*
  2. *Programming recursive functions on flat lists.*
- *Deadline: Week 6, Thursday November 05, 4.00pm.*
- *Electronic submission: Electronic submission only. Four files:*
  1. `diagonal.scm`*: answer to question 1.*
  2. `average.scm`*: answer to question 2.*
  3. `set-union.scm`*: answer to question 3.*
  4. `graphs.scm`*: answer to question 4.*
- *Learning outcomes:*
  1. *to program in a functional style*
  2. *the key mechanisms underpinning the functional programming model*
  3. *understand the concept of functional programming and be able to write programs in this style in the context of Scheme.*
- *Scheme Language: Standard (R5RS)*
- *Marking scheme: 25% for each question.*
- *Automated marking: Forty percent (40%) of the marks allocated to each question will be computed* ***automatically*** *by running a test sequence based on the illustrations given in appendix. The rest of the marks will be be based on style (e.g. code structure and layout, choice of recursion schema, efficiency). Note that tests in appendix are purely illustrative: a correct solution is expected to satisfy these tests; however, satisfying a test sequence is not a guarantee of a correct solution. The automated marks will be determined by the tests in appendix and further tests to ensure that solutions are not hard-coded for a specific test sequence. Syntax of the notation used in the test sequence and details of how to run tests yourself are available from* `https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/testing.html`

1. We consider a matrix representation as a list of rows, each row itself being represented as a list. All rows are expected to be of the same length, but the matrix is not expected to be square (it may have different numbers of rows and columns). For instance, a matrix of $m+1$ rows and $n+1$ columns is represented as follows.

$$\begin{array}{cccccc} ((& a_{00} & a_{01} & \ldots & a_{0n} &) \\ (& a_{m0} & a_{m1} & \ldots & a_{mn} &)) \end{array}$$

   The diagonal of a matrix is the list of entries $(a_{00}, \ldots, a_{ij}, \ldots)$ where $i = j$.

   Write a function `diagonal` that takes a matrix and returns its diagonal.

   Appendix A contains illustrations of the function `diagonal`.

2. Let us consider a list of numbers $(a_0, \ldots, a_n)$. For each prefix of such a list, we can compute its average.

prefix	average
(a_0)	$a_0/1$
$(a_0\ a_1)$	$(a_0 + a_1)/2$
$(a_0\ a_1\ a_2)$	$(a_0 + a_1 + a_2)/3$
$\vdots$	
$(a_0\ a_1\ \ldots\ a_n)$	$(a_0 + a_1 + \ldots + a_n)/(n+1)$

   Define a function `increasing-average` that takes a list of numbers and returns `#f` if the average ot its successive prefixes is sometimes decreasing, and returns the average of the whole list if the average of successive prefixes is never decreasing.

   A correct solution will avoid unecessary floating point operations.

   Appendix B contains illustrations of the function `increasing-average`.

3. We define the "listset" datatype as an implementation of sets that relies on linked lists for their representation. We assume the existence of the following primitives.

`(listset-add e s)`	adds an element `e` to a listset `s`
`listset-empty`	the empty listset
`(listset-null? s)`	predicate to check if a listset `s` is empty
`(listset-first s)`	accessor to return the first element of a listset `s` (implementation chooses which)
`(listset-rest s)`	accessor to return the remaining elements of a listset `s` (all but the one that would be returned by listset-first)
`(listset-equals? s1 s2)`	predicate to compare two listsets `s1` and `s2`.

   Define a function `listset-union` that takes two listsets and computes their union.

   Make sure that your solution relies on the primitives above to manipulate listsets. To facilitate the testing of your solution, the test file contains an implementation of the primitives. To gain full marks your solution must not be dependent on the concrete implementation of listsets. Your code will be tested against alternative implementation of the listset datatype.

   Appendix C contains illustrations of the function `listset-union` as well as immplementation of the listset primitives.

4. We represent a directed graph by a list of edges, each edge being represented by a list of two elements consisting of a source node and a destination node (See Figure 1).

   Conceptually, a node map is a partial function that maps nodes to nodes. A node map is represented as an association list.

   Applying a node map m to a graph G_1 results in a new graph G_2, such that:

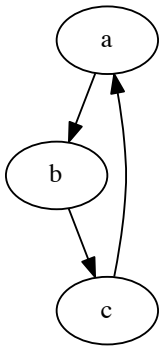
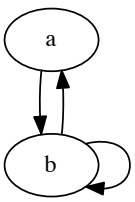


Figure 1: Example of directed graph: `( (a b) (b c) (c a) )`

- Each node of G_1 present in m is mapped to a node in G_2.
- Nodes of G_1 that are not mapped by m are ignored.
- An edge e of G_1 is mapped to an edge in G_2 if both the source and destination nodes of e have been mapped to G_2.

Figure 2 provide 3 different maps of the graph illustrated in Figure 1.

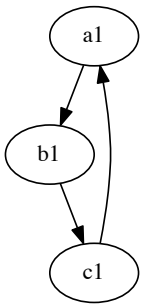
node merging



map:

```
(   (a a)
    (b b)
    (c b) )
```

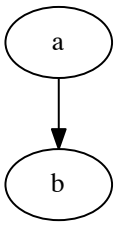
node renaming



map:

```
(   (a a1)
    (b b1)
    (c c1) )
```

node dropping



map:

```
(   (a a)
    (b b) )
```

Figure 2: Examples of node maps applied to the graph of Figure 1

Define a function `graph-mapper` that takes a graph and a map represented as described above and returns a new graph, which is the result of applying the node map to the input graph.

Appendix D contains illustrations of the function `graph-mapper`.

A Illustrations for `diagonal`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk1/diagonal-public.tst

```
(load "diagonal.scm")
+++

(diagonal '((10)))
$diagonal-1
(10)

(diagonal '((1 2) (3 4)))
$diagonal-2
(1 4)

(diagonal '((1 2 3) (a b c) (4 5 6)))
$diagonal-3
(1 b 6)

(diagonal '())
$diagonal-0
() ;; base case, empty matrix


(diagonal '((1 2 3) (a b c)))
$diagonal-not-square0
(1 b)

(diagonal '((1 2) (a b) (foo far)))
$diagonal-not-square1
+++
```

B Illustrations for `increasing-average`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk1/average-public.tst

```
(load "average.scm")
+++

(increasing-average '()) ;;; error: no average
***

(increasing-average '(-2))
-2

(increasing-average '(-2 -5))
#f

(increasing-average '(2 2 2)) ;;; remaining constant
2

(increasing-average '(2 3 4 5)) ;;; numbers are increasing
7/2

(increasing-average '(2 3 4 5 4)) ;;; numbers are not increasing, but average is
18/5

(increasing-average '(2 3 4 5 4 3)) ;;; decreasing average
#t

(increasing-average '(2 3 4 5 4 3.7))
3.6166666666666667
```

C Illustrations for `listset-union`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk1/set-union-public.tst

```
(load "set-union.scm")
+++

(begin
  (define listset-add
    (lambda (element set)
      (if (member element set)
          set
          (cons element set))))


  (define listset-empty '())
  (define listset-null?
    (lambda (set)
      (equal? set listset-empty)))

  (define listset-first car)
  (define listset-rest cdr)

  (define listset-included?
    (lambda (set1 set2)
      (if (listset-null? set1)
          #t
          (if (member (listset-first set1) set2)
              (listset-included? (listset-rest set1) set2)
              #f))))

  (define listset-equal?
    (lambda (set1 set2)
      (and (listset-included? set1 set2)
           (listset-included? set2 set1)))))
---


(begin
  (define set1 (listset-add '1 listset-empty))
  (define set2 (listset-add '2 set1))
  (define set3 (listset-add '3 set2))

  (define seta (listset-add 'a listset-empty))
  (define setb (listset-add 'b seta))
  (define setc (listset-add 'c setb))
  (define setd (listset-add 'd setc)))
---

(listset-equal? seta set3)
#f

(listset-equal? (listset-union set3 set3) set3)
```

```
#t

(listset-equal? (listset-union setd setd) setd)
#t

(listset-equal? (listset-union set3 setd) (listset-union setd set3))
#t

(listset-equal? (listset-union setd listset-empty) setd)
#t

(listset-equal? (listset-union listset-empty setd) setd)
#t

(listset-equal? (listset-union listset-empty listset-empty) listset-empty)
#t
```

D Illustrations for `graph-mapper`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk1/graph-public.tst

```
(load "graph.scm")
+++

(begin
  (define *map1* '((a . a) (b . b) (c . b)))
  (define *map2* '((a . a1) (b . b1) (c . c1)))
  (define *map3* '((a . a) (b . b)))
  (define *map4* '((x . y)))
  (define *map5* '())

  (define *graph1*  '((a b) (b c) (c a)))
  )
---

(graph-mapper *graph1* *map1*)
((a b) (b b) (b a))

(graph-mapper *graph1* *map2*)
((a1 b1) (b1 c1) (c1 a1))

(graph-mapper *graph1* *map3*)
((a b))

(graph-mapper *graph1* *map4*)
()

(graph-mapper *graph1* *map5*)
()
```