Programming III COMP2209

Coursework 1

Individual Coursework *(cf. collaboration policy)*

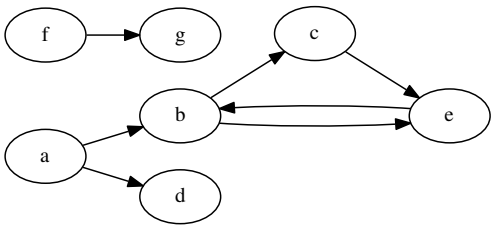
- *Aims of this coursework:*
  1. *Programming in the DrRacket environment.*
  2. *Programming recursive functions on flat lists and binary trees.*
- *Deadline: Week 6, Thursday November 07, 4.00pm.*
- *Electronic submission: Five files:*
  1. `node-degree.scm`*: answer to question 1.*
  2. `group-by-degree.scm`*: answer to question 2.*
  3. `make-csv.scm`*: answer to question 3.*
  4. `graph-inverse.scm`*: answer to question 4.*
  5. `tree-list.scm`*: answer to question 5.*
- *Marking scheme: 20% for each question.*

The purpose of questions 1 to 4, in this coursework, is to compute the degree distribution of a graph, or more precisely a degree histogram. Even though the questions are concerned with graphs, they only require flat recursion.

One can easily use association lists to represent graphs, associating each node with the nodes it is connected to (or adjacent to). We assume that nodes are simply represented as symbols. The accessor function `successors`, defined below, returns the list of adjacent nodes in a graph (this is sometimes referred to as adjacency list).

Oct 25: We assume that all nodes of the graph are represented explicitly in the association list, including those without successors, such as `g` and `d`.

```
(define *graph*
  '((a . (b d))
    (b . (c e))
    (c . (e))
    (d . ())
    (e . (b))
    (f . (g))
    (g . ())))
```



```
(define successors
  (lambda (node graph)
    (let ((val (assq node graph)))
      (if val
          (cdr val)
          '()))))
```

In the study of graphs (see `http://en.wikipedia.org/wiki/Degree_distribution`), the degree of a node in a graph is defined as the number of connections it has to other nodes; the degree distribution is the probability distribution of these degrees over the whole graph. A degree histogram displays the frequency (the number of nodes that have a given degree) for each degree.

The degree of a node in a graph is the number of edges the node has to other nodes. If a graph is directed, meaning that edges point in one direction from one node to another node, then nodes have two different degrees, the in-degree, which is the number of incoming edges, and the out-degree, which is the number of outgoing edges.

1. Define a function `node-degree` that takes a graph as input and computes the out-degree of each node in the graph. The result is also an association list associating each node with its out-degree.

   ```
   (define *graph*
     '((a . (b d))
       (b . (c e))
       (c . (e))
       (e . (b))
       (f . (g))))

   (node-degree *graph*)
      -> ((a . 2) (b . 2) (c . 1) (d . 0) (e . 1) (f . 1) (g . 0))
   ```

   The order in which elements are returned is not specified. Oct 25: note that degree for `d` and `g` is 0.

2. Define a function `group-by-degree` that takes an association list associating graph nodes with their out-degree, and returning an association list that associates each degree with the nodes that have that out-degree.

   ```
   (group-by-degree (node-degree *graph*))
      -> ((2 a b) (1 c e f) (0 d g))
   ```

   Oct 25: nodes associated with 0 are `d` and `g`.

   Again, the order in which degrees are listed is not specified, nor the order of nodes for a given degree.

3. Define a function `make-csv` that takes a graph and returns a string, containing comma-separated values (usually abbreviated CSV) for each degree and frequency.

   A call to `(make-csv *graph*)` returns the following string:

   ```
   #Degree, Frequency
   0, 2
   1, 3
   2, 2
   ```

A line is used for each degree. The first line is starting with # to indicate a comment, specifying the name of each column in this csv file. Oct 25: note that there is a line for degree 0.

For this question, marks will be allocated as follows:

- 40%: overall functionality of `make-csv`
- 30%: reusing functions defined in 1) and 2)
- 30%: sorting the results, making use of the `list-sort` function[1], defined as follows.

```
(define insert
  (lambda (less-than x lst)
    (if (null? lst)
        (list x)
        (let ((y (car lst))
              (ys (cdr lst)))
          (if (less-than x y)
              (cons x lst)
              (cons y (insert less-than x ys)))))))

(define list-sort
  (lambda (less-than lst)
    (if (null? lst)
        '()
        (insert less-than
                (car lst)
                (list-sort less-than (cdr lst))))))
```

4. The inverse of a graph contains the same nodes, but for each edge (x, y) in the original, we find an edge (y, x) in the inverse. Define the function `graph-inverse` that takes a graph and computes its inverse. The inverse graph is also represented as an association list.

```
(graph-inverse *graph*)
  ->((b a e) (d a) (c b) (e b c) (g f) (a) (f))
```

Oct 25: note that `a` and `f` are nodes without successors in the inverse graph.

For this question, 30% of the marks will be allocated for solutions that define common procedures shared by functionality defined in previous examples.

5. We consider the following definition for binary trees:

   (a) A binary tree is either a node or an empty tree:

   $$\langle btree \rangle \quad ::= \quad \langle node \rangle \mid \langle empty\ tree \rangle$$

   (b) A *node* results from the construction of a value and two binary trees by the constructor `make-node`, which has the following signature.

   $$\texttt{make-node} \quad : \quad \langle value \rangle \times \langle btree \rangle \times \langle btree \rangle \rightarrow \langle node \rangle$$

   (c) An `empty tree` is created by the constructor `make-empty-tree`, which has the following signature.

   $$\texttt{make-empty-tree} \quad : \quad void \rightarrow \langle empty\ tree \rangle$$

[1] Note that Scheme defines a sort function in the standard r6rs library, but their implementations in DrRacket and Bigloo used in comp2209 appear to be incompatible. Instead make use of the sort function provided here.

An example of such trees is as follows.

```
(define *tree*
  (make-node 'a
             (make-node 'b
                        (make-empty-tree)
                        (make-empty-tree))
             (make-node 'c
                        (make-node 'd
                                   (make-empty-tree)
                                   (make-empty-tree))
                        (make-node 'e
                                   (make-empty-tree)
                                   (make-empty-tree)))))
```

The abstract data type binary tree can be implemented by the following record types:

```
(define-record-type node
    (make-node v l r)
    node?
    (v tree-value)
    (l tree-left)
    (r tree-right))

(define-record-type empty-tree
    (make-empty-tree)
    empty-tree?)
```

Define a function `tree->list` that takes a binary tree and returns an association list, associating each node label, with the labels of nodes that are directly children.

```
(tree->list *tree*)
   -> ((a b c) (b) (c d e) (d) (e))
```