CSE1729 - Introduction to Programming — April 3, 2017

Prelim 2 Solutions

1. Pairs and Lists. You may use the SCHEME built-in function `append` without defining it if you wish to use it in the following problems.
   A binary relation, R, from a set A to a set B is a set of ordered pairs where the first element of each ordered pair comes from A and the second element comes from B. For example, if $A = \{1,2,3\}$, $B = \{0,2,4\}$ and R represents the "less than" relation (e.g. $a < b$), then $R = \{(1,2),(1,4),(2,4),(3,4)\}$. Using SCHEME lists to represent sets, we can define functions that determine the relations between two sets.

   (a) [3 points] Define a SCHEME function, `(in-relation value R B)` that takes a `value`, a function `R` and a list `B` and returns a list of pairs of the value with all of the values `b` in the list `B` such that *value R b* is true. For example, using the set B above, `(in-relation 1 < B)` evaluates to the list `'((1 . 2) (1 . 4))`.

   Solution:

```
(define (in-relation value R B)
  (if (null? B)
      (list)
      (if (R value (car B))
          (cons (cons value (car B)) (in-relation value R (cdr B)))
          (in-relation value R (cdr B)))))
```

   (b) [4 points] Using your `in-relation` function from part a, define a SCHEME function, `(relation-set A R B)`, which takes a list representing the set A, a function R, and a list representing the set B as parameters and evaluates to the list of pairs representing the relation R. For example, using the sets A and B above and the "less than" relation, `(relation-set A < B)` should return `'((1 . 2) (1 . 4) (2 . 4) (3 . 4))`.

   Solution:

```
(define (relation-set A R B)
  (if (null? A)
      (list)
      (append (in-relation (car A) R B)
              (relation-set (cdr A) R B))))
```

   (c) [3 points] The inverse of a relation R, denoted by R^{-1} is the set of all pairs `(b . a)` such that the pair `(a . b)` is in the relation R. Define a SCHEME function named `(inverse-relation R)` that takes a list of pairs, `R`, as a parameter, and evaluates to another list of pairs in which the first and second elements of each pair in `R` are reversed. For example, if `R` is the list `'((1 . 2) (1 . 4) (2 . 4) (3 . 4))`,

then `(inverse-relation R)` evaluates to
`'((2 . 1) (4 . 1) (4 . 2) (4 . 3))`.

Solution:

```
(define (inverse-relation R)
  (if (null? R)
      (list)
      (cons (cons (cdar R) (caar R)) (inverse-relation (cdr R)))))
```

2. Lists and Trees

(a) [3 points] Define a SCHEME function named `delete-val` that takes two parameters, a value `v` and a list of integers `l`. This function should return the list of the values in list `l` with all occurrences of the value `v` removed.

Solution:

```
(define (delete-val v l)
  (cond ((null? l) (list))
        ((eq? v (car l)) (delete-val v (cdr l)))
        (else (cons (car l) (delete-val v (cdr l))))))
```

(b) [3 points] Using your `delete-val` function from part a, define a SCHEME function named `delete-dups` that takes one parameter that is a list of integers and returns a list of integers with no duplicate values. It should, of course, preserve the first occurrence of every duplicate value in the list as well as the values that are not duplicated. For example,

```
>(delete-dups (list 1 2 3 2 4 2 5 2 6 6 7 3 8 4 5 2 6 7 8 9))
'(1 2 3 4 5 6 7 8 9)
```

Solution:

```
(define (delete-dups l)
  (if (null? l)
      (list)
      (cons (car l) (delete-dups (delete-val (car l) (cdr l))))))
```

(c) [4 points] Define a SCHEME function, `(flatten l)`, that takes a **list** of values, and other lists as a parameter and returns a single list containing all of the values in the original list as elements (e.g. removed from their pairs and lists). For example,flattening the following tree:

```
(define tree '(1 (2 (3 (4 () ()) (5 (6 () ()) ()))
        (7 (8 () ()) (9 (10 () ()) ()))) (11 (12 (13 () ())
        (14 (15 () ()) ()))(16 (17 () ())(18 (19 () ()) ()))))))
(flatten tree)
```

returns the list `'(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)`
(Hint: Be sure to include the case where an empty list is an element of a list.)

Solution:

```
(define (flatten l)
  (cond ((null? l) l)
        ((null? (car l)) (flatten (cdr l)))
        ((pair? (car l)) (append (flatten (car l))
                                 (flatten (cdr l))))
        (else (append (list (car l)) (flatten (cdr l))))))
```

3. Binary Search Trees (BST)

   (a) [4 points] Define a SCHEME function, `(num-leaves T)`, which takes one parameter that is a Binary Search Tree and returns the number of leaf nodes in T.

Solution:

```
(define (num-leaves T)
  (cond ((null? T) 0)
        ((and (null? (left T)) (null? (right T))) 1)
        (else (+ (num-leaves (left T)) (num-leaves (right T))))))
```

   (b) [6 points] Write a SCHEME function which, given a binary search tree T and an integer z, returns the number of integers in the tree that are greater than or equal to z. For example, given the tree below and the number 5, your function should return 4, since there are 4 numbers in the tree that are greater than or equal to 5 (specifically, 5, 8, 11, and 21).
   Your solution should exploit the fact that the tree is a binary search tree to avoid considering certain subtrees.

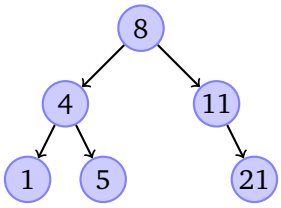

Figure 1: A binary search tree with 6 nodes, 4 of which are 5 or larger.

Solution:

```
(define (count-larger T z)
  (cond ((null? T) 0)
        ((<= z (value T))
         (+ 1 (count-larger (left T)) (count-larger (right T))))
```

```
        ((> z (value T))
         (count-larger (right T)))))
```

4. The following problems concern the heap data structure. You may use the convenience functions defined in lecture (`h-min`, `insert`, `remove-min`, and `combine-heaps`) without defining them yourself.

   (a) [5 points] Give a SCHEME function `heap-equal` which, given two heaps H_1 and H_2, returns `#t` if the two heaps contain the same elements (not necessarily with the same structure), and `#f` otherwise.

   Solution:

```
(define (heap-equal H1 H2)
  (cond ((and (null? H1) (null? H2)) #t)
        ((null? H1) #f)
        ((null? H2) #f)
        ((= (value H1) (value H2))
           (heap-equal (remove-min H1)
                       (remove-min H2)))
        (else #f)))
```

   (b) [5 points] Recall that a *heap* is a binary tree with the property that *the value of any node is less than (or equal to) that of its children*.

   Write a SCHEME function which, given a heap H and an integer z, returns a heap that contains all elements of H that are smaller than z.

   Solution:

```
(define (trim-heap H z)
  (cond ((null? H) H)
        ((< (value H) z) (make-tree
                           (value H)
                           (trim-heap (left H) z)
                           (trim-heap (right H) z)))
        (else '())))
```