CSE 1729 - Introduction to Principles of Programming November 11, 2015

Prelim 2

This is a 50 minute exam, commencing at 11:15am and finishing at 12:05pm. You may not use any reference material during the exam, including books or notes. You may not use any calculators, computers, or cell phones during the exam. *Please read every question carefully before answering* and express your answers as completely as you can.

There are 4 questions on the exam, each worth 10 points; **you may choose *any three* to complete**: The highest score that can be achieved on the exam is 30 points. Please indicate which 3 questions you wish to be graded in the check boxes next to the question numbers below.

Name: ______________________________

Question	Grade?	Score
1		
2		
3		
4		
Total		

Tree Conventions: A Refresher

Problems 1, 3, and 4 on the exam concern binary trees and heaps. As in lecture, we consider binary trees with the property that each node stores—in addition to its two subtrees—an integer. Recall the convention that we used for such trees: each node is represented by a list

```
(v left right),
```

where `v` is a value, and `left` and `right` are its two subtrees.

Please adopt this convention in your responses. Make free use of the "convenience" functions:

```
(define (make-tree v left-tree right-tree)
   (list v left-tree right-tree))
(define (value T) (car T))
(define (left T) (cadr T))
(define (right T) (caddr T))
```

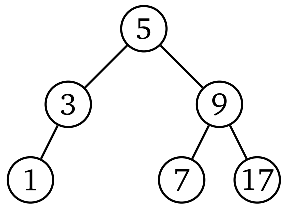
The first of these functions builds a tree from a value and a pair of trees; the following three functions extract, from a tree T, the value of the root and its left and right subtrees. Finally, we used the Scheme empty list to represent the empty tree.

Heaps Recall that a *heap* is a binary tree with the *heap property*: any node's value is no less that of its parent. (Equivalently, any node's value is less than (or equal to) that of its children.) The convenience functions (as defined in class)

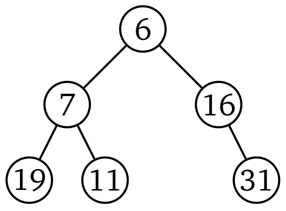
```
(define (create-heap v H1 H2) (list v H1 H2))
(define (h-min H) (car H))
(define (left H) (cadr H))
(define (right H) (caddr H))
```

can be used to define and manipulate heaps. Recall that `h-min` retrieves the smallest element at the top of a heap whereas `left` and `right` retrieve the left and right sub-heaps of a heap H.

Binary search trees A *binary search tree* is a tree with the property that all nodes in the "left-hand" tree beneath a given node have values *smaller* than the node; likewise, all nodes in the "right-hand" subtree beneath a node have values *larger* than the node.



A binary search tree



A heap

1. Operations on lists and trees.

   (a) (3 points) Define a Scheme function named `filter` which takes a function, `f` and a list, `lst`, as parameters and returns a list composed of all the elements `x` of `lst` for which `(f x)` returns `#t`. The order of elements in the result should be the same as in the original list.

   Solution:

   ```
   (define (filter  f lst)
     (cond ((null? lst) lst)
           ((f (car lst))
            (cons (car lst)(filter f (cdr lst))))
           (else
            (filter f (cdr lst)))))
   ```

   (b) (3 points) Define a Scheme function `(max-over gf lst)` which takes a function of two variables, `gf` and a list, `lst`, as parameters and returns the maximum value in that list as defined by that function. The function `gf` is used to compare two values: if `(gf x y)` returns `#t`, then we consider `x` to be greater than y. The use of the Scheme function `apply` is not allowed here.

   Solution:

   ```
   (define (max-over lst fun)
     (define (max a b)
       (if (fun a b) a b))
     (if (null? (cdr lst))
         (car lst)
         (max (car lst)(max-over (cdr lst) fun))))
   ```

   (c) (4 points) We define `(nest-depth x)` as follows: if `x` is a non-empty list, `(nest-depth x)` is 1 plus the maximum nest-depth of its elements; if `x` is the empty list, or not a list, then `(nest-depth x)` is 0. Define a SCHEME function `nest-depth` which calculates `(nest-depth x)` for any `x`. Feel free to use `max-over` as a helper function.

   Solution:

   ```
   (define (nest-depth lst)
     (cond ((null? lst) 0)
           ((not (list? lst)) 0)
           (else
            (+ 1 (max-over (map nest-depth lst) >)))))
   ```

2. This question is concerned with list processing.

   (a) (3 points) Define a SCHEME function `suffix-sum` which, given a list $(a_1\ a_2\ \ldots\ a_k)$ of integers, returns the list $(s_1\ s_2\ \ldots\ s_k)$ where s_i is the sum of the first i of the a_i. Specifically,

$$s_1 = a_1 + a_2 + \cdots + a_k, \quad s_2 = a_2 + \cdots + a_k, \quad \ldots, \quad s_k = a_k.$$

   You may use helper functions, but you may not use `let` or `let*`. For full credit you should not do any unnecessary additions.

Solution:

```
(define (suffix-sum lst)
  (define (sum-and-cons x lst)
    (cons (+ x (car lst)) lst))
  (cond ((null? lst) lst)
        ((null? (cdr lst)) lst)
        (else
         (sum-and-cons (car lst)
                       (suffix-sum (cdr lst))))))
```

   (b) (3 points) Write a Scheme function `(subset-of-size?  lst1 lst2 n)` which returns `#t` if 1) `lst1` represents a set (that is, it has no repeats among its elements), 2) it is a subset of `lst2`, and 3) it has `n` elements. You can assume all elements of `lst1` and `lst2` are symbols.

Solution:

```
(define (subset-of-size? lst1 lst2 n)
  (cond ((and (null? lst1)(= n 0)) #t)
        ((or (null? lst1)(= n 0)) #f)
        ((and (member (car lst1) lst2)
              (not (member (car lst1) (cdr lst1))))
         (subset-of-size? (cdr lst1) lst2 (- n 1)))
        (else  #f)))
```

   (c) (4 points) Write the Scheme function `all-choices lst n` which evaluates to a list containing all of the subsets of `lst` that have `n` elements. For example, `all-choices '(a b) 2)` could evaluates to `((a b)(a c)(b c))`. You may assume that all elements are symbols; the order of elements within the result and in the individual subsets is unimportant (so `((c b)(a c)(b a))` would have been an equally good answer to the above example.

Solution:

```
(define (all-choices lst n)
  (cond ((= n 0) '(()))
        ((null? lst) lst)
        ((append
          (map (lambda(x)(cons (car lst) x))
               (all-choices (cdr lst) (- n 1)))
          (all-choices (cdr lst) n)))))
```

3. Recall that a *binary search tree* is a binary tree with the property that *the value at the root is greater than all values in its left subtree and less than all values in its right subtree.*

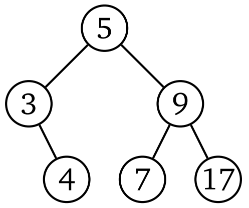
   (a) (4 points) Write a SCHEME function `bst-insert` which, given a number x and a binary search tree B, returns a binary search tree that contains x and all of the elements of B (without repeats).

Solution:

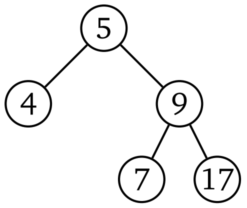
```
(define (bst-insert val bst)
  (cond ((null? bst)
          (make-tree val '() '()))
        ((= val (value bst))
          bst)
        ((< val (value bst))
          (make-tree (value bst)
                     (bst-insert val (left bst))
                     (right bst)))
        (else
          (make-tree (value bst)
                     (left bst)
                     (bst-insert val (right bst))))))
```

   (b) (6 points) Write a SCHEME function `bst-gteq` which, given a binary search tree B and an number z, returns a binary search tree that contains all elements of B that are greater to or equal to z.

   For example, given the binary search tree on the left below and the number 4, your function could return the binary search tree on the right. This function should take advantage of the structure of the tree, and *not* work by successive remove-min's.



B



(bst-gteq B 4)

Solution:

```
(define (bst-gteq bst z)
  (cond ((null? bst) bst)
        ((> z (value bst))
          (bst-gteq (right bst) z))
        (else
          (make-tree (value bst)
                     (bst-gteq (left bst) z)
                     (right bst)))))
```

4. 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.* Be sure to use the heap convenience functions from page 2 in solving these.

   (a) (5 points) Write a SCHEME function `heap-max` which, given a heap h, returns the maximum value stored in that heap.

Solution:

```
(define (heap-max h)
  (cond ((null? h) 'undefined)
        ((and (null? (left h))(null? (right h)))
         (h-min h))
        ((null? (left h)) (heap-max (right h)))
        ((null? (right h)) (heap-max (left h)))
        (else (max (heap-max (left h))(heap-max (right h))))))
```

   (b) (5 points) Give a SCHEME function `(heap-insert val h)` which, given a value `val` and a heap `h`, returns the heap that results from inserting `val` into heap `h`.

Solution:

```
(define (heap-insert val heap)
  (cond ((null? heap)
         (create-heap val '() '()))
        ((< val (h-min heap))
         (create-heap val
                      (right heap)
                      (heap-insert (h-min heap)(left heap))))
        (else
         (create-heap (h-min heap)
                      (right heap)
                      (heap-insert val (left heap))))))
```