CSE1729 - Introduction to Programming November 7, 2016

Prelim 2

This is a 50 minute exam, commencing at 10:10am and finishing at 11:00am. 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 3 and 4 on the exam concern binary trees and heaps. As in class, 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. You may make free use of the "convenience" functions:

```
(define (maketree 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))
```

Heaps Recall that a *heap* is a binary tree with the *heap property*: any node's value is at least 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 an 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.

1. Pairs and Lists

(a) (**3 points**) Consider the problem of *mapping* the values in a list, namely, given a function f and a list $\ell = (v_0, v_1, \cdots, v_k)$, compute the list $(f(v_0), f(v_1), \cdots, f(v_k))$. Write a SCHEME function `map` which takes as input a function f of one argument and a list ℓ and returned a list of all the elements of ℓ transformed by f.

Solution:

```
(define (map f l)
  (cond ((null? l) '())
        (else (cons (f (car l)) (map f (cdr l))))))
```

(b) (**3 points**) Armed with a `map` function, write a `pairup` function which, given a single value v and a list ℓ computes a list of pairs between v and the elements of ℓ. The call `(pairup 1 (list 'a 'b 'c))` should produce `'((1 . a) (1 . b) (1 . c))`.

Solution:

```
(define (pairup v l)
  (map (lambda (x) (cons v x)) l))
```

(c) (**4 points**) Now that you have `pairup`, write a `cross` function which, given a pair of lists ℓ_1 and ℓ_2 computes a list of all the possible pairings between elements of ℓ_1 and ℓ_2. For instance, a call `(cross (list 1 2 3) (list 'a 'b))` would produce the list of pairs `'((1 . a) (1 . b) (2 . a) (2 . b) (3 . a) (3 . b))`.

Solution:

```
(define (cross a b)
  (cond ((null? a) '())
        (else (append (pairup (car a) b)
                      (cross (cdr a) b)))))
```

2. Lists

(a) (**5 points.**) Define a SCHEME function named `rotate-left` that takes a list as a parameter and returns a list obtained by taking the first element of the original list and placing it at the end of the list returned by the function. For instance:

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

Solution:

```
(define (rotl lst)
  (define (rotl-aux x lst)
    (if (null? lst)
        (list x)
        (cons (car lst) (rotl-aux x (cdr lst)))))
  (rotl-aux (car lst) (cdr lst)))
```

(b) (**5 points.**) Define a SCHEME function named `rotate-right` that takes a list as a parameter and returns a list obtained by taking the last element of the original list and placing it at the front of the list returned by the function. For instance:

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

Solution:

```
(define (rotr l)
  (if (null? (cdr l))
      (cons (car l) (list))
      (let ((end-list-pair (rotr (cdr l))))
        (cons (car end-list-pair) (cons (car l) (cdr end-list-pair))))))
```

3. (**10 points.**) Write a SCHEME function `check-tree` which, given a binary tree, returns `#t` if the tree has the structure of a binary search tree, and `#f` otherwise. (We say that a binary tree "has the structure of a binary search tree" if the value at any node is greater than or equal to the values in the left subtree, and less then or equal to the values in the right subtree.) **Hint.** If it is helpful, you may assume that all numbers in the tree lie between —1000 and 1000. One way to get started is to write a function `all-smaller` so that `(all-smaller T x)` returns `#t` if all elements of T are less than or equal to x; likewise, you could define an analogous function `all-larger`.

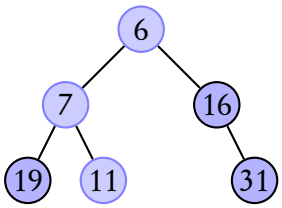
Solution:

```
(define (check-tree T)
  (define (bst-with-values T a b)
    (cond ((null? T) #t)
          ((and (<= a (value T))
                (<= (value T) b))
           (and (bst-with-values (left T)
                                 a
                                 (value T))
                (bst-with-values (right T)
                                 (value T)
                                 b)))
          (else #f)))
  (bst-with-values T -1000 1000))
```

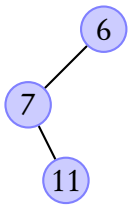
4. (**10 points.**) Recall that a *heap* is a binary tree with the property that *the value of any node is less than (or equal to) that of its children*.

   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.

   For example, given the heap on the left in Figure 1 below and the number 15, your function could return the heap on the right hand side.



Before deletion



After deletion

Figure 1: On left, a heap. On right, the remaining heap after all elements greater or equal to 15 have been removed.

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 '())))
```

Scratch Space