CSE1729 - Introduction to Programming | April 3, 2017

Prelim 2 Solutions

1. (**10 points.**) Define SCHEME functions with the following specifications.

   (a) (**1 points.**) Give a SCHEME function `reverse` so that `(reverse p)`, given a pair p, returns the pair with the coordinates reversed. For example `(reverse (cons 1 2))` should return the pair `(2 . 1)`.

```
(define (reverse p)
  (cons (cdr p) (car p)))
```

   (b) (**3 points.**) Give a SCHEME function `listlookup` so that `(listlookup L k)` returns the kth element of the list $L = (\ell_1 \ldots \ell_n)$. For example, `(listlookup '(1 3 5 7) 2)` should return 3. (You may just assume that the parameter k will be "in-range"—you don't need to gracefully handle the case where there is no kth element.)

```
(define (listlookup L k)
  (if (= k 1)
      (car L)
      (listlookup (cdr L) (- k 1))))
```

   (c) (**3 points.**) Consider the following function for computing the maximum element of a non-empty list of numbers:

```
(define (listmaximum numbers)
  (cond ((null? (cdr numbers)) (car numbers))
        ((>= (car numbers) (listmaximum (cdr numbers)))
         (car numbers))
        (else (listmaximum (cdr numbers)))))
```

   The function is correct, but there are some cases where it is *very* slow. In particular, the invocation `(listmaximum '(1 2 3 ... 1000))` will not terminate in your lifetime. Explain the problem and explain precisely how to fix it.

   The problem is that `listmaximum`, when called on a list of length n, could generate as many as two recursive calls to `listmaximum` on a list of length $n-1$. This can result in $\approx 2^n$ recursive calls, which grows very quickly as a function of n. To fix the problem, restructure the code so that it makes only one recursive call and remembers the result. For example:

```
(define (listmaximum numbers)
  (if (null? (cdr numbers))
      (car numbers)
      (let ((tailmax (listmaximum (cdr numbers))))
         (if (> (car numbers) tailmax)
             (car numbers)
             tailmax))))
```

   (d) (**3 points.**) Define a SCHEME function `cubes` so that `(cubes k)` returns a list of the first k cubes in ascending order. Thus `(cubes 3)` should return the list `(1 8 27)`. (If you want to use a list `reverse` function, that's fine, but you need to define it from scratch.)

```
(define (cubes k)
  (define (cube-iter a)
     (if (> a k)
         '()
         (cons (* a a a) (cube-iter (+ a 1)))))
  (cube-iter 1))
```

2. (**10 points.**) Recall the notion of *binary search tree*.

   (a) (**5 points.**) Define a SCHEME function `insert` so that `(insert v S)`, given a binary search tree `S` and a number `v`, properly inserts `v` into `S`; your function should return the resulting tree.

```
(define (bst-insert v T)
  (cond ((null? T) (maketree v '() '()))
        ((< v (value T)) (maketree (value T)
                                   (bst-insert v (left T))
                                   (right T)))
        (else            (maketree (value T)
                                   (left T)
                                   (bst-insert v (right T))))))
```

   (b) (**3 points.**) Define a SCHEME function `max` so that `(max S)`, given a non-empty binary search tree S, outputs the largest element in S.

```
(define (max S)
  (if (null? (right S))
      (value S)
      (max (right S))))
```

   (c) (**1 points.**) One simple implementation of the SET abstract datatype uses an (unordered) list to maintain the set. A problem with this implementation is that you may have to traverse the entire list to determine membership.

   We originally introduced the binary search tree data structure as a technique for more efficiently maintaining a set. However, there are circumstances where it can be just as bad as a list. Explain how this can happen, and what the tree looks like.

   If, for example, the elements of the set are inserted in sorted order the resulting tree will consist of a single path consisting of the elements in sorted order. In particular, the depth of the tree will be equal to the number of inserted elements.

   (d) (**1 points.**) However, there are circumstances where the binary search tree can be extremely efficient. In the best case, how many values can you pack in to a binary search tree of depth d?

   The complete binary tree of depth d contains exactly $2^{d+1}-1$ nodes. In particular, the tree has exactly 2^d leaves.

3. (**10 points.**) Counting small values in trees. Recall the definitions of trees, heaps, and binary search trees.

   (a) (**3 points.**) Define a SCHEME function `count-smaller` so that `(count-smaller T k)`, given a tree T and an integer k, returns the number of nodes in T which contain values less than k. Note that `count-smaller` *must work for all trees: it may not assume that T is a heap or binary search tree.*

```
(define (count-smaller T k)
  (if (null? T)
      0
```

```
      (+ (count-smaller (left T) k)
         (count-smaller (right T) k)
         (if (< (value T) k) 1 0))))
```

(b) (**4 points.**) Define a SCHEME function `heap-count-smaller` so that `(heap-count-smaller H k)`, given a heap H and an integer k, returns the number of nodes in H which contain values less than k. Of course, since a heap is a binary tree, you could use the "generic" code you gave in 3a. For credit, your solution must use the special structure of a heap to give an optimized solution that does not necessarily need to traverse the entire tree.

```
(define (heap-count-smaller H k)
  (cond ((null? H) 0)
        ((< (value H) k) (+ (heap-count-smaller (left H) k)
                            (heap-count-smaller (right H) k)
                            1))
        (else 0)))
```

(c) (**2 points.**) Consider the same problem, but given a binary search tree S. Again, it would be possible to use the generic code of 3a to count the number of values in S less than a particular value k. Explain (in English) how to use the binary search tree structure to give an optimized solution that does not necessarily need to traverse the entire tree.

Note that if the value of a particular node in a binary search tree is *larger than or equal to* k, only the left (smaller) subtree needs to be explored for smaller elements. This leads to the following function:

```
(define (bst-count-smaller S k)
  (cond ((null? S) 0)
        ((< (value S) k) (+ (bst-count-smaller (left S) k)
                            (bst-count-smaller (right S) k)
                            1))
        (else (bst-count-smaller (left S) k))))
```

(d) (**1 points.**) Which structure—binary search tree or heap—is preferable for this particular problem? Explain why.

In fact, heaps have a (small) advantage over binary search trees in this setting. For heaps, the number of nodes visited by the function will be directly proportional to the number of nodes that hold values less than k (in fact, the algorithm will not visit more than three times as many nodes as there are that hold values less than k). On the other hand, regardless of the circumstances, the binary search tree algorithm will have to descend through the tree to visit the node associated with the smallest element.

4. (**10 points.**) Define a SCHEME function `sort` so that `(sort L)`, given a list L of integers, returns them (as a new list) in sorted order.

```
(define (quicksort elements)
  (if (null? elements)
      elements
      (let ((pivot (car elements))
            (rest (cdr elements)))
        (define (partition remainder smaller larger)
          (if (null? remainder)
              (cons smaller larger)
              (if (< (car remainder) pivot)
                  (partition
                    (cdr remainder)
```

```
                (cons (car remainder) smaller)
                larger)
              (partition
                (cdr remainder)
                smaller
                (cons (car remainder) larger)))))
      (let ((pieces (partition rest '() '())))
        (append (quicksort (car pieces))
                (list pivot)
                (quicksort (cdr pieces)))))))
```