Quiz 5b Rubric

1. (3 points) Louis Reasoner has run into a problem reconciling mutual recursion with data abstraction. He says that in several mutual recursion procedures, there are conceptually two base cases – the base case for a forest (the “horizontal” base case) and the base case for a tree (the “vertical” base case), but only one base case needs to be written – a check for the empty list in the forest helper procedure. The problem that he found was that the “vertical” base case has to do with the tree ADT, and so checking for this base case using null? is a data abstraction violation. Is Louis right? If yes, explain how we could change the procedure sum-tree so that this problem doesn’t happen. If no, explain the error in Louis’ reasoning.

(define (sum-tree tree)

(+ (datum tree)

(sum-forest (children tree))))

(define (sum-forest forest)

(if (null? forest)

0

(+ (sum-tree (car forest))

(sum-forest (cdr forest)))))

No, Louis is wrong. The “vertical” base case is when a tree has no children. The ADT guarantees that the children of a tree will be a list of trees itself, and so it is not a data abstraction violation to use null? on the children.

1 point for No.

2 points for explanation. Note: If they said Yes, and then said that they would create a special empty tree variable, and check for that instead of the empty list, give them 1 point.

1. (3 points) Write the procedure max-of-tree which finds the maximum number in a tree of nodes. This is the general tree ADT, which has the constructor make-tree and selectors datum and children. For full credit, your procedure must work on trees with negative numbers.

(define (max-of-tree tree)

(if (null? (children tree))

(datum tree)

(max (datum tree) (max-of-forest (children tree)))))

(define (max-of-forest forest)

(if (null? (cdr forest))

(max-of-tree (car forest))

(max (max-of-tree (car forest)) (max-of-forest (cdr forest)))))

(define (max-of-tree tree)

(accumulate max (datum tree)

(map max-of-tree (children tree))))

Use your own judgment for partial credit. Either mutual recursion or accumulate-map is fine.

1. (4 points) Fill in the rest of the permutations procedure, which takes in a list with **no repeated elements** and returns a list of all possible permutations of that list. Here’s the algorithm:
   1. For all of the numbers in the list:
      1. For all of the permutations in the list without that number:
         1. Consing the number with the permutation gives one permutation of the output.

> (permutations ‘(4 7 2))

((4 7 2) (4 2 7) (7 4 2) (7 2 4) (2 4 7) (2 7 4))

(define (flatmap f lst)

(accumulate append ‘() (map f lst)))

(define (permutations lst)

(if (or (null? lst) (null? (cdr lst)))

(list lst)

(flatmap (lambda (num)

(map (lambda (perm)

(cons num perm))

(permutations

(filter (lambda (x) (not (= x num)))

lst))))

lst)))

Assign partial credit using your best judgment. Some things to look for when giving partial credit – the recursive call, the call to filter, appending num to perm using cons, making sure that the second map operates on the result of the recursive call, etc.