Fake Plastic Trees

QUESTIONS

1. Write \((\text{square-tree} \ \text{tree})\), which returns the same tree structure, but with every element squared. Don’t use “map”!

   \[
   \begin{align*}
   \text{(define (square-tree tree)} \\
   &\quad \text{(make-tree (square (datum tree)))} \\
   &\quad \text{(square-forest (children tree))))
   \end{align*}
   \]

   \[
   \begin{align*}
   \text{(define (square-forest forest)} \\
   &\quad \text{(if (null? forest)} \\
   &\quad \quad \text{'}()) \\
   &\quad \text{(cons (square-tree (car forest)) (square-forest (cdr forest)))))
   \end{align*}
   \]

2. Write \((\text{max-of-tree} \ \text{tree})\) that does the obvious thing. The tree has at least one element.

   \[
   \begin{align*}
   \text{(define (max-of-tree tree)} \\
   &\quad \text{(if (null? (children tree)} \\
   &\quad \quad \text{(datum tree)} \\
   &\quad \quad \text{(max (datum tree) (max-of-forest (children tree)))))}
   \end{align*}
   \]

   \[
   \begin{align*}
   \text{(define (max-of-forest forest)} \\
   &\quad \text{(if (null? (cdr forest)} \\
   &\quad \quad \text{(max (max-of-tree (car forest)) (max-of-forest (cdr forest)))))}
   \end{align*}
   \]

3. Write \((\text{listify-tree} \ \text{tree})\) that turns the tree into a list in any order.

   \[
   \begin{align*}
   \text{(define (listify-tree tree)} \\
   &\quad \text{(cons (datum tree) (listify-forest (children tree)))))}
   \end{align*}
   \]

   \[
   \begin{align*}
   \text{(define (listify-forest forest)} \\
   &\quad \text{(if (null? forest)} \\
   &\quad \quad \text{'}()) \\
   &\quad \text{(append (listify-tree (car forest)) (listify-forest (cdr forest)))))}
   \end{align*}
   \]

4. A maximum heap is a tree whose children’s data are all less-than-or-equal-to the root’s datum. Of course, its children are all maximum heaps as well. Write \((\text{max-heap?} \ \text{tree})\) that checks if a given tree is a maximum heap.

   \[
   \begin{align*}
   \text{(define (max-heap? tree)} \\
   &\quad \text{(and (= (datum tree) (max-of-tree tree))} \\
   &\quad \quad \text{(max-heaps? (children tree))))))
   \end{align*}
   \]

   \[
   \begin{align*}
   \text{(define (max-heaps? forest)} \\
   &\quad \text{(cond ((null? forest) #t)} \\
   &\quad \quad \text{(else (and (max-heap? (car forest))} \\
   &\quad \quad \quad \text{(max-heaps? (cdr forest)))))))}
   \end{align*}
   \]
Binary Search Trees

QUESTIONS

1. Write \( \text{sum-of bst} \) that takes in a binary search tree, and returns the sum of all the data in the tree.

\[
\text{(define (sum-of bst)}
\begin{align*}
\text{(cond ((null? bst) 0)} \\
\text{else (+ (datum bst)} \\
\text{ (sum-of (left-branch bst)))} \\
\text{ (sum-of (right-branch bst))})
\end{align*}
\]

2. Write \( \text{max-of bst} \) that takes in a binary search tree, and returns the maximum datum in the tree. The tree has at least one element. (Hint: This should be easy.)

\[
\text{(define (max-of bst)}
\begin{align*}
\text{(cond ((null? (right-branch bst)) (datum bst))} \\
\text{(else (max-of (right-branch bst)))})
\end{align*}
\]

3. Write \( \text{remove-leaves bst} \) that takes in a bst and returns the bst with all the leaves removed.

\[
\text{(define (remove-leaves bst)}
\begin{align*}
\text{(cond ((null? bst) '())} \\
\text{((leaf? bst) '())} \\
\text{else (make-tree (datum bst)} \\
\text{ (remove-leaves (left-branch bst)))} \\
\text{ (remove-leaves (right-branch bst))})
\end{align*}
\]

4. Write \( \text{height-of tree} \) that takes in a tree and returns the height – the length of the longest path from the root to a leaf.

\[
\text{(define (height-of tree)}
\begin{align*}
\text{(cond ((leaf? tree) 0)} \\
\text{else (+ 1 (max (height-of (left-branch tree))} \\
\text{ (height-of (right-branch tree))))})
\end{align*}
\]

5. \text{(HARD!) Write \( \text{width-of tree} \) that takes in a tree and returns the width – the length of the longest path from one leaf to another leaf.}

\[
\text{(define (width-of tree)}
\begin{align*}
\text{(cond ((or (null? tree) (leaf? tree)) 0)} \\
\text{else (+ 2 (max (height-of (left-branch tree))} \\
\text{ (height-of (right-branch tree))} \\
\text{ (width-of (left-branch tree))} \\
\text{ (width-of (right-branch tree))}))}
\end{align*}
\]

6. Jimmy the Smartass was told to write \( \text{valid-bst? bst} \) that checks whether a tree satisfies the binary-search-tree property – elements in left subtree are smaller than datum, and elements in right subtree are larger than datum. He came up with this:

\[
\text{(define (valid-bst? bst)}
\begin{align*}
\text{(cond ((null? bst) #t)} \\
\text{else (and (or (null? (left-branch bst))} \\
\text{ (and (< (datum (left-branch bst)) (datum bst))} \\
\text{ (valid-bst? (left-branch bst))}} \\
\text{ (or (null? (right-branch bst))})}
\end{align*}
\]
Why will Jimmy never succeed in life? Give an example that would fool his pitiful procedure.

Checking if the bst property is true for your immediate children’s labels does not guarantee that the property holds for the whole subtree. For example, this tree would fool valid-bst?:

```
  10
 / \
 5 18
 / \
 1 30
```

The 1 violates the bst property (1 is not larger than 10), but Alex’s algorithm will merely check that 1 is smaller than 18, and move on.

Can you do better?

7. Write (listify bst) that converts elements of the given bst into a list. The list should be in NON-DECREASING ORDER!

```
(define (listify bst)
  (cond ((null? bst) '())
    (else (append (listify (left-branch bst))
                   (list (datum bst))
                   (listify (right-branch bst))))))
```

Deep Lists

QUESTIONS

1. Write deep-sum, that takes in a deep-list, and returns the sum of every element in the deep-list.

```
> (deep-sum '(1 (2 3) (4 (5) 6) (7 (8 9))))
> 45
```

```
(define (deep-sum ls)
  (cond ((null? ls) 0)
        ((atom? (car ls))
         (+ (car ls) (deep-sum (cdr ls)))))
        (else (+ (deep-sum (car ls))
                 (deep-sum (cdr ls))))))
```

2. Write a procedure replace-with-depth, that takes in a deep-list, and returns the same list structure, but with each element replaced by its depth in the list

```
> (hello (my name (is)) garply)
> (1 (2 2 (3)) 1)
```
(define (replace-with-depth ls)
  (define (replace-helper ls counter)
    (cond ((null? ls) '())
          ((atom? (car ls))
           (cons counter (replace-helper (cdr ls) counter)))
          (else (cons (replace-helper (car ls) (+ counter 1))
                      (replace-helper (cdr ls) counter))))
    (replace-helper ls 1))

3. Write deep-accumulate, that works like accumulate, but on deep-lists:

   > (deep-accumulate * 1 '(1 2 3 4 (5 (6) (7 8))))
   > 40320

   (define (deep-accumulate op initial lst)
    (cond ((null? lst) initial)
          ((atom? lst) lst)
          (else (op (deep-accumulate op initial (car lst))
                     (deep-accumulate op initial (cdr lst))))))

; alternatively here, you could write a procedure to flatten a deep-list, and then just call accumulate on the flattened list.