CSE1729 – Introduction to Programming | March 28, 2017

Problem Set 7 Solutions

Recall the conventions we have adopted in class for maintaining trees. We represent the empty tree with the empty list `()`; a nonempty tree is represented as a list of three objects

```
(value left-subtree right-subtree)
```

where `value` is the value stored at the root of the tree, and `left-subtree` and `right-subtree` are the two subtrees. We introduced some standardized functions for maintaining and accessing this structure, which we encourage you to use in your solutions below. (Use the SCHEME `eq?` function to test equality for the values of the nodes.)

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

1. Define a SCHEME procedure, named `tree-equal?`, which takes two trees as parameters and returns true if the trees are identical (same values in the same places with the same structure) and false otherwise.

```
(define (tree-equal Ta Tb)
  (cond ((and (null? Ta) (null? Tb)) #t)
        ((or  (null? Ta) (null? Tb)) #f)
        (else (and (eq? (value Ta) (value Tb))
                   (tree-equal (left  Ta) (left  Tb))
                   (tree-equal (right Ta) (right Tb))))))
```

2. Define a SCHEME procedure, named `tree-sort`, which takes a list of numbers and outputs the same list, but in sorted order. Your procedure should sort the list by

   (a) inserting the numbers into a binary search tree and, then,

   (b) extracting from the binary search tree a list of the elements in sorted order.

   To get started, write a procedure called `insert-list` which takes a list of numbers L and a tree T, and returns the tree that results by inserting all numbers from L into T.

   Then write a function called `sort-extract` which takes a binary search tree and outputs the elements of the tree in sorted order. (We did this in class!)

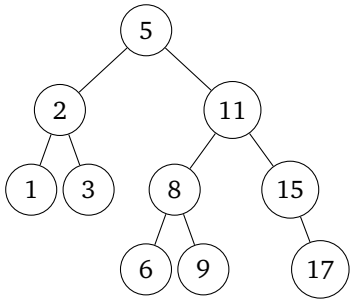
   Then, finally, put these two functions together to achieve `tree-sort`.

```
(define (tree-sort elements)
  (define (make-tree value left right) (list value left right))
  (define (value tree) (car tree))
  (define (left tree)  (cadr tree))
  (define (right tree) (caddr tree))
  (define (insert element T)
    (cond ((null? T) (make-tree element '() '()))
          ((< element (value T)) (make-tree (value T)
                                            (insert element (left T))
                                            (right T)))
          (else (make-tree (value T)
```

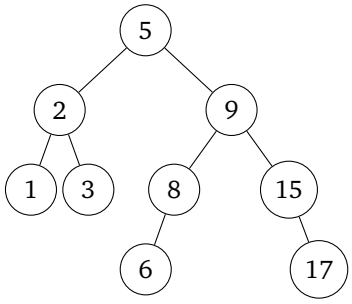
```
                          (left T)
                          (insert element (right T))))))
(define (insert-list insert-elements T)
  (if (null? insert-elements)
      T
      (insert-list (cdr insert-elements)
                   (insert (car insert-elements) T))))
(define (sort-extract T)
  (if (null? T)
      '()
      (append (sort-extract (left T))
              (list (value T))
              (sort-extract (right T)))))
(sort-extract (insert-list elements '()))
```

3. Define a SCHEME procedure, named `delete-node` that takes a binary search tree, `T`, and a value, `v`, as parameters and returns a binary search tree with the node that contains the value `v` removed from the tree. Note that the tree your function returns *must maintain the binary search tree property*. Removing interior (that is, non-leaf) nodes requires a little care. (See Example 1 for an example.)

Deleting nodes from trees The `delete-node` function is tasked with removing the one occurrence of v from the input binary search tree T. Naturally, if T has n nodes, the output tree T' has $n-1$ nodes and T' satisfies the binary search tree property (*for every node n, the nodes in the left sub-tree of n hold values smaller than the value held in n and the nodes in the right-subtree of n hold values larger than the value held in n*). There are number of cases that need to be handled separately: the node n may have no sub-trees, exactly one subtree, or two subtrees. (See Example 3.) The first two situations can be handled easily. The third case, illustrated below in Example 2, requires that you restructure the tree a bit to reattach the two "orphaned" subtrees of n in an appropriate way. One can either replace the value at the deleted node with the largest value in the left subtree or the smallest value in the right subtree while removing the corresponding leaf node.

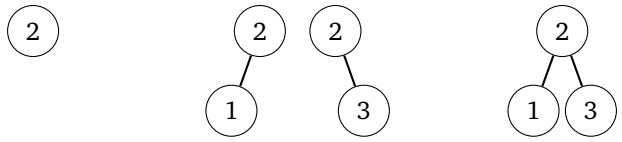


Example 1: An example of a binary search tree.



Example 2: The same binary search tree after the removal of the value 11. Note, we chose to promote the largest value in the left subtree (9) to the node vacated by the value 11.

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



Example 3: There are three possibilities when finding a node to remove. (The middle two cases are treated as one.)

```
(define (right tree) (caddr tree))
(define (delete v T)
  (define (leaf? T) (and (null? (left T))
                         (null? (right T))))
  (define (tree-max T) (if (null? (right T))
                           (value T)
                           (max (value T)
                                (tree-max (right T)))))

  (cond ((null? T) '())
        ((< v (value T)) (make-tree (value T)
                                    (delete v (left T))
                                    (right T)))
        ((> v (value T)) (make-tree (value T)
                                    (left T)
                                    (delete v (right T))))
        ((null? (left T)) (right T))
        (else (make-tree (tree-max (left T))
                         (delete (tree-max (left T))
                                 (left T))
                         (right T)))))
```

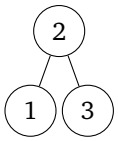
What follows is an alternate version that traverses the left subtree only once to produce both the maximum element and the subtree with the element removed.

```
(define (bst-find-delete v T)
 (define (is-leaf? T)
   (and (null? (left T)) (null? (right T))))
 (define (bst-find-delete-max T)
   (if (is-leaf? (right T))
       (cons (value (right T))
             (make-tree (value T) (left T) (list)))
       (let ((val-tree (bst-find-delete-max (right T))))
         (cons (car val-tree)
               (make-tree (value T) (left T) (cdr val-tree))))))
 (cond ((is-leaf? T) (if (equal? v (value T))
                         (list)
                         T))
   ((equal? v (value T))
    (if (not (null? (left T)))
        (let ((get-max (bst-find-delete-max (left T))))
          (list (car get-max) (cdr get-max) (right T)))
        (right T)))
```

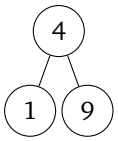
```
   ((< v (value T)) (make-tree (value T)
                               (bst-find-delete v (left T))
                                                  (right T)))
   (else (make-tree (value T)
                    (left T) (bst-find-delete v (right T))))))
```

4. *SICP Exercise 2.31.* Define a SCHEME procedure, named `tree-map`, which takes two parameters, a tree, `T`, and a function, `f`, and is analogous to the `map` function for lists. Namely, it returns a new tree T' with a topology identical to that of T but where each node $n \in T'$ contains the image under f of the value stored in the corresponding node in T. For instance, if the input tree is the tree shown in Example 4 and the function `f` is $f(x) = x^2$. then the tree returned by `tree-map` is shown in Example 5.

```
(define (tree-map T f)
  (if (null? T)
      '()
      (make-tree (f (value T))
                 (tree-map (left T) f)
                 (tree-map (right T) f))))
```



Example 4: T, the tree passed to `tree-map`



Example 5: T', the tree returned by `tree-map` when passed T and the squaring function as parameters.

5. This problem concerns ways to represent arithmetic expressions using trees. For this purpose, we will consider 4 arithmetic operations: $+$ and $*$, both of which take two arguments, and $-$ and $1/\square$, both of which take one argument. (As an example of how these one-argument operators work, the result of applying the operator $-$ to the number 5 is the number -5; likewise, the result of applying the $1/\square$ operator to the number 5 is the number $1/5$.)

   An *arithmetic parse tree* is a special tree in which every node has zero, one, or two children and:

   - each leaf contains a numeric value, and
   - every internal node with exactly two children contains one of the two arithmetic operators $+$ or $*$,
   - every internal node with exactly one child contains one of the two arithmetic operators $-$ or $1/\square$.

   (You may assume, for this problem and the next, that when a node has a single child, this child appears as the left subtree.)

   If T is an arithmetic parse tree, we associate a value with T (which we call value(T)) by the following recursive rule:

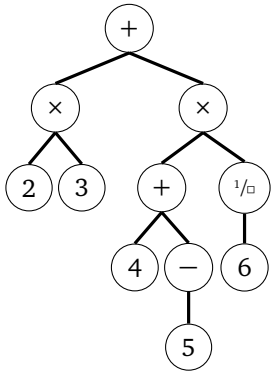
   - if T has a single (leaf) node, value(T) is equal to the numeric value of the node,
   - if T has two subtrees, L and R, and its root node contains the operator $+$, then value(T) = value(L) + value(R),

- if T has two subtrees, L and R, and its root node contains the operator $*$, then value(T) = value$(L) *$ value(R),
- if T has one subtree, S, and its root node contains the operator $-$, then value$(T) = -$value(S),
- if T has one subtree, S, and its root node contains the operator $^1/_\square$, then value$(T) = 1/$ value(S).

You can see how to associate with any arithmetic expression a natural arithmetic parse tree. Note that since $-$ and $^1/_\square$ are *unary* operators (take only one argument), you have to give a little thought to how to represent expressions such as $3-5$ or $3/5$. For example, the arithmetic parse tree for the expression

$$2 \times 3 + \frac{4-5}{6}$$

is shown in Example 6.



Example 6: An arithmetic parse tree for the expression $2 \times 3 + (4 + (-5)) \times (1/6)$.

Write a SCHEME function which, given an arithmetic parse tree, computes the value associated with the tree. You may assume that the operators +, ×, −, and $^1/_\square$ appear in the tree as the *characters* #\+, #*, #\-, and #\/. (See the comments at the end of the problem set concerning the SCHEME character type.) To test your code, try it out on the parse tree

```
(define example (list #\+ (list #\*
                                (list 4 '() '())
                                (list 5 '() '()))
                      (list #\+
                            (list #\/ (list 6 '() '()) '())
                            (list 7 '() '()))))
```

The solution:

```
(define (nvalue T)
  (cond ((eq? (value T) #\+) (+ (nvalue (left T))
                                (nvalue (right T))))
        ((eq? (value T) #\*) (* (nvalue (left T))
                                (nvalue (right T))))
        ((eq? (value T) #\-) (- (nvalue (left T))))
        ((eq? (value T) #\/) (/ 1 (nvalue (left T))))
        (else (value T))))
```

6. (A continuation of the previous problem; pre-, in-, and post-order traversal.) Such trees can be traversed recursively (which you will have done in the solution to your previous problem). In this problem, you will print out the expression associated with a parse tree, using several different conventions for writing arithmetic expressions.

   There are three conventional orders in which nodes and subtrees can be "visited" during a recursive traversal of a tree. Note that at each node, there three tasks to carry out: visit (in this case, that means *print out*) the node, traverse the left subtree, and traverse the right subtree. For instance, an *inorder* traversal of a binary tree will:

   1) recursively traverse the left subtree,
   2) visit the node, and then
   3) recursively traverse the right subtree.

   Therefore, an inorder traversal yields *infix notation*. Likewise, a preorder traversal visits each node first and then recursively traverses the left and then the right subtrees. A preorder traversal produces the expression in *prefix notation* as shown in Example 7. And, yes, as you may have guessed, a postorder traversal examines the left subtree, then the right subtree and finally visits the root node itself. A postorder traversal produces the same expression in postfix notation. The postfix notation for the example expression is "$23\times45-+6\frac{1}{\square}\times+$".

   (a) Define a SCHEME function that takes a binary expression tree and uses a preorder scan on the tree to produce the expression in prefix (also called "Polish") notation. Your function should return a *string* containing the expression in prefix notation. See the following section regarding characters and strings in SCHEME.

   One difficulty is that your tree has values of different "types"—the leaves have numbers, the internal nodes have characters. You will need to convert everything to strings. You could do this by hand, or you can map the following function on your tree (using your previous solution to `tree-map`).

```
(define (prepare x)
  (cond ((number? x) (number->string x))
        ((char? x) (string x))))
```

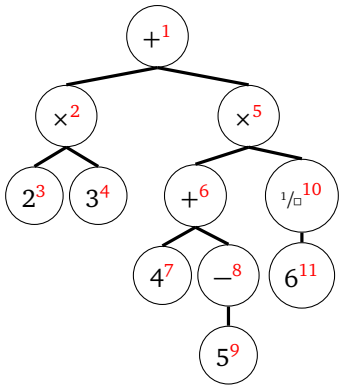
   This function will convert all characters and numbers to strings, so that your tree contains only string labels.

   Once the tree contains only strings, you can build the final expression by appending the strings together to build your final expression.

   (Note that the function `number?` returns true if its argument is a number; likewise, the function `char?` returns true if its argument is a character. The function `number->string` converts a number to a string.)

```
(define (prefix T)
  (define (prefix-aux T)
   (if (null? T)
       ""
       (string-append (value T) (prefix-aux (left T))
                                (prefix-aux (right T)))))
  (prefix-aux (tree-map T prepare)))
```

   (b) Define a SCHEME function that takes a binary expression tree as a parameter and uses a postorder traversal to produce a string representing the expression in postfix notation (also called "Reverse Polish Notation," or RPN). After the previous problem, this is easy! Incidentally, several early computer and calculator architectures were designed with RPN. Several current languages such as PostScript and software programs including the MacOSX calculator, the Unix dc calculator program and several Android and iPhone apps make use of RPN.



Example 7: An arithmetic parse tree showing "preorder" numbering of the nodes (in red). This gives the expression in prefix notation as "$+ \times 23 \times +4 - 5\frac{1}{\square}6$."

```
(define (postfix T)
  (define (postfix-aux T)
    (if (null? T)
        ""
        (string-append (postfix-aux (left T))
                       (postfix-aux (right T))
                       (value T))))
  (postfix-aux (tree-map T prepare)))
```

(c) It is an interesting fact that when the numbers are only a single digit long, the postfix expression associated with an arithmetic tree *completely determines the tree*; that same is true for the prefix expression. This is not true for infix expressions: "2+3*6" can be generated from two different arithmetic tree (which given different values!). Give an algorithm which converts a tree into an infix expression but adds parentheses around the arguments of every $*$, $\times$, $-$, and $\frac{1}{\square}$. To be more precise, a $\times$ or $+$ operator produces output of the form $(a_1 \times a_2)$ (or $(a_1 + a_2)$), whereas the operators $-$ and $\frac{1}{\square}$ produce output of the form $-(a_1)$ (or $/(a_1)$). Thus, your algorithm, when applied to the tree pictured above, should yield the string "$((2 \times 3) + ((4 + -(5)) \times (/(6))))$." Note that the parentheses do uniquely determine the tree from which the expression arose.

```
(define (infix T)
  (cond ((eq? (value T) #\+)
         (string-append "(" (infix (left T)) "+"
                            (infix (right T)) ")"))
        ((eq? (value T) #\*)
         (string-append "(" (infix (left T)) "*"
                            (infix (right T)) ")"))
        ((eq? (value T) #\-)
         (string-append "-(" (infix (left T)) ")"))
        ((eq? (value T) #\/)
         (string-append "1/(" (infix (left T)) ")"))
        (else (number->string (value T)))))
```

Strings and characters in SCHEME SCHEME has the facility to work with strings and characters (a *string* is just a sequence of characters). In particular, SCHEME treats *characters* as atomic objects that evaluate to themselves. They are denoted: `#\a`, `#\b`, ... Thus, for example,

```
> #\a
#\a
> #\A
#\A
> (eq? #\A #\a)
#f
> (eq? #\a #\a)
#t
```

The "space" character is denoted `#\space`. A "newline" (or carriage return) is denoted `#\newline`.

A *string* in SCHEME is a sequence of characters, but the exact relationship between strings and characters requires an explanation. A string is denoted, for example, `"Hello!"`. You can build a string from characters by using the `string` command as shown below. An alternate method is to use the `list->string` command, which constructs a string from a list of characters, also modeled below. Likewise, you can "explode" a string into a list of characters by the command `string->list`:

```
> (string #\S #\c #\h #\e #\m #\e)
"Scheme"
> (list->string '(#\S #\c #\h #\e #\m #\e))
"Scheme"
> (string->list "Scheme")
(#\S #\c #\h #\e #\m #\e)
> "Scheme"
"Scheme"
```

Finally, given two strings, you can append them together using the `string-append` function (alternatively, you could turn them in to lists, append those, and convert back):

```
> (string-append "book" "worm")
"bookworm"
> (list->string (append (string->list "book") (string->list "worm")))
"bookworm"
```

Note that strings, like characters, numbers, and Boolean values, evaluate to themselves.