CSE 1729	Introduction to Principles of Programming Problem Set 7	October 20, 2017

In this problem set you will write code for binary trees. 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.

```
(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. 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:

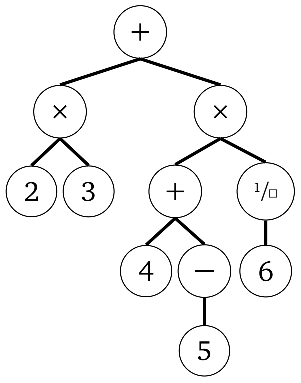
   - each leaf contains a numeric value, and
   - every internal node with exactly two children contains one of the two arithmetic operators $+$ or $*$,
   - every internal node with exactly one child contains one of the two arithmetic operators $-$ or $^1/_\square$.

   (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 1.



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

Write a SCHEME function, named `(nvalue T)`, which, given an arithmetic parse tree, computes the value associated with the tree. You may assume that the operators $+$, $\times$, $-$, 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 '() '()))))
```

2. (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 2. 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 “$23 \times 45 - +6\frac{1}{\square} \times +$”.

(a) Define a SCHEME function, named `(prefix T)`, 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.

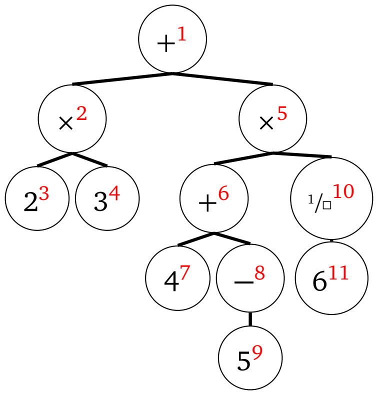
One difficulty is that your tree has values of different “types”—the leaves have numbers, the internal nodes have characters. You will need to convert everything to strings. You could do this by hand, or you can map the following function on your tree (using your previous solution to `tree-map`).

```
(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.)



Example 2: An arithmetic parse tree showing “preorder” numbering of the nodes (in red). This gives the expression in prefix notation as “$+ \times 23 \times +4 - 5\frac{1}{\square}6$.”

(b) Define a SCHEME function, named `(postfix T)`, 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.

(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 give different values!). Give an function, named `(infix T)`, which converts a tree into an infix expression but adds parentheses around the arguments of every $*$, $\times$, $-$, and $\frac{1}{\Box}$. 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 $\frac{1}{\Box}$ produce output of the form $-(a_1)$ (or $/(a_1)$). Thus, your function, 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.

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.

3. In lab this week, you wrote some functions to work with binary trees. In this question, you will develop some code for *binary search trees*, as we have been doing in lecture. The representation of a binary search tree will be the same as the binary trees we used in lab, that is, a tree is a three-element list: the value of the node, the left subtree, and the right subtree, with empty trees represented as the empty list.

   Unlike the binary search trees we have been discussing in lecture, for the questions here we will assume all of the values are strings, so you will need to use the comparison functions `string<?` , `string>?`, and `string=?` functions to build and search through your trees.

   Note: For all of the binary search tree problems in 3 and 4, you should work with the data in the tree (or trees), and *not* extract the data from the tree into a list and solve the problem on the list. While that strategy could work, you would lose the advantages of having the data in a binary search tree, and will not get credit for these *even if the code passes the Mimir tests*.

   (a) Using your tree functions, write a Scheme function `(bst-element? item bs-tree)` that evaluates to `#t` if `item` is the value of a node in `bs-tree`, `#f` otherwise.

   (b) Using these functions, write a Scheme function `(bst-insert item bs-tree)` that evaluates to the binary search tree that that results from inserting the value `item` into binary search tree `bs-tree`. Remember that we do not allow duplicate values in our binary search trees.

   (c) Write the Scheme function `(bst-smallest bs-tree)` that evaluates to the smallest value in binary search tree `bs-tree`. If there are no values in the tree, it should evaluate to `'undefined`).

   (d) Write the Scheme function `(bst-largest bs-tree)` that evaluates to the largest value in binary search tree `bs-tree`. If there are no values in the tree, it should evaluate to `'undefined`).

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

   (f) Since these binary search trees contain strings (their node values) without repeats, they are ideal for representing sets of strings. Write a Scheme function `(bst-subset? bst1 bst2)` that evaluates to `#t` if the set of values in `bst1` is a subset of the set of values in `bst2`, `#f` otherwise.

   (g) Use `bst-subset` to write a Scheme function `(bst-set-equal? bst1 bst2)`, that evaluates to `#t` if the set of values in `bst1` is the same as the set of values in `bst2`, `#f` otherwise.

4. Deleting a value from a binary search tree can be tricky – when you delete a node containing a value, the result must still be a binary search tree. There is an explanation below of the various cases you will need to consider. You will do this in parts, starting with the easier cases and then solving the general problem.

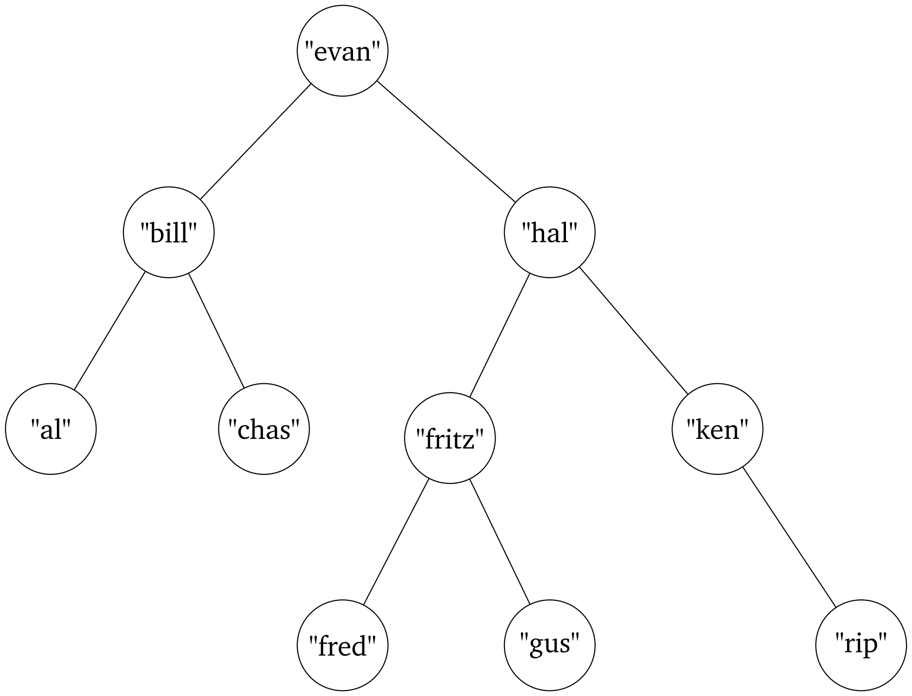
   For each of these delete functions, the trees that are returned *must maintain the binary search tree property* explained below.

(a) Define a Scheme procedure, named `bst-delete-min` that takes a binary search tree, `bst` as a parameter and returns a binary search tree with the node that contains the minimum value removed from the tree – that is, a binary search tree with the same values except for the minimum one. This procedure should *not* use `bst-smallest` or `bst-delete`.

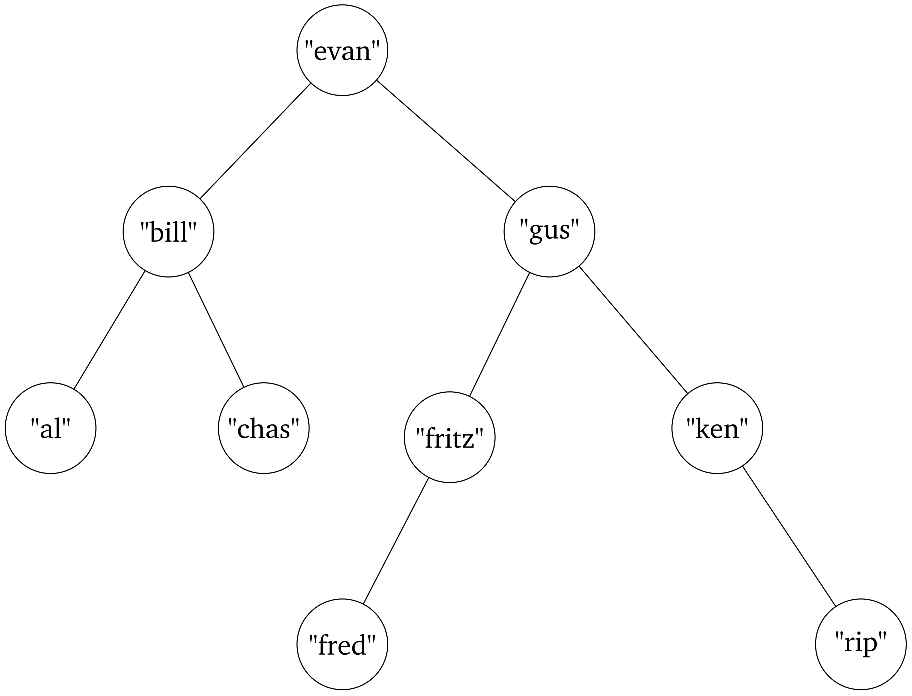
(b) Define a Scheme procedure, named `bst-delete-max` that takes a binary search tree, `bst` as a parameter and returns a binary search tree with the node that contains the maximum value removed from the tree. This procedure should *not* use `bst-largest` or `bst-delete`.

(c) Finally, define a Scheme procedure, `(bst-delete val bst)` that takes a number `val` and a binary search tree `bst` as parameters and returns a binary search tree with the node that whose value is equal to `val` removed from the tree – that is, the values in the resultant binary search tree will be the same as the original binary search tree with the exception of `val`; if `val` is in the original tree's set it will not be in the resultant tree's set, but the tree will have the same set if `val` was not in the original set.

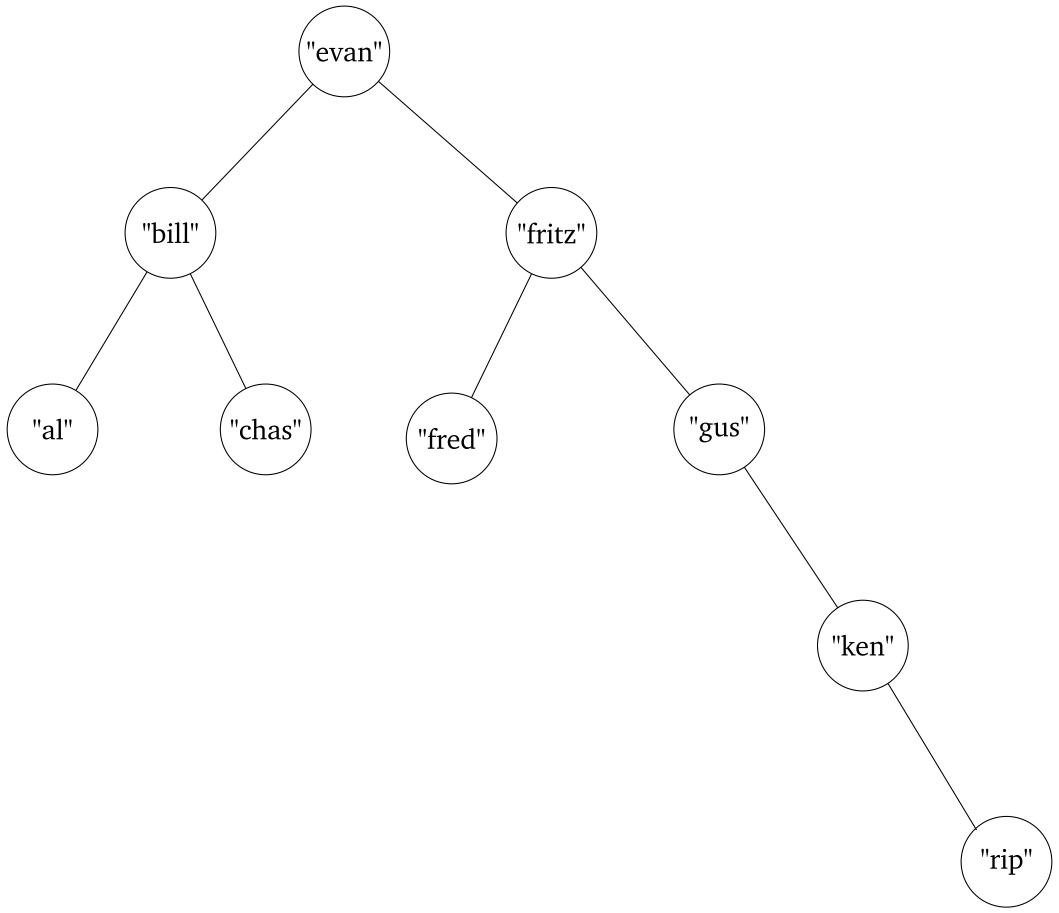
Deleting nodes from trees Each of the above delete functions is tasked with removing one value (and its node) from the input binary search tree `bst`. Naturally, if `bst` has n nodes, the output tree has $n-1$ nodes (unless the value to be removed is not in the tree) and satisfies the binary search tree 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 6.) The first two situations can be handled easily (Hint: how do these two relate to `bst-delete-min` and `bst-delete-max`?). The third case, illustrated below in Example 4, requires that you restructure the tree a bit to reattach the two "orphaned" subtrees of n in an appropriate way. One can either 1) 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, or 2) promote the left subtree to the place where the node is to be deleted, and connect the right subtree to the node with the maximum value in the left subtree (you could also promote the right subtree and connect the left subtree to the right's minimum-valued node).



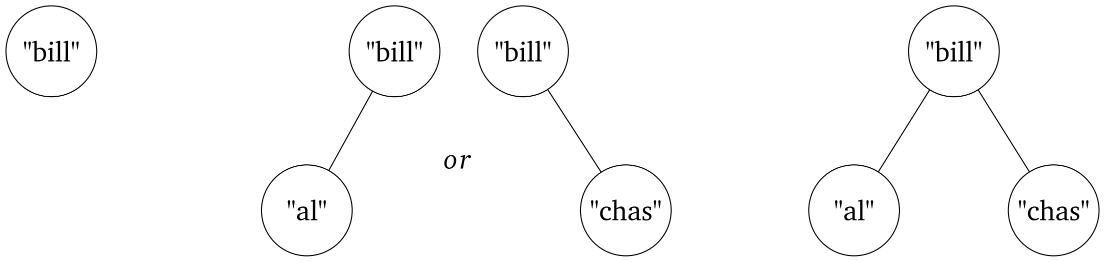
Example 3: An example of a binary search tree.



Example 4: The same binary search tree after the removal of the value `"hal"`. Note, we chose to promote the largest value in the left subtree (`"gus"`) to the node vacated by the value `"hal"`.



Example 5: The same binary search tree after the removal of the value `"hal"`. Note, we chose to promote the left subtree, and connected the right subtree to the node with the largest value in the left subtree `"gus"`.



Example 6: There are three possibilities when finding a node to remove. For each of these we are removing the node containing `"bill"`.