COMP 212 Spring 2022
Homework 07

In lecture and lab, we looked at higher-order functions on lists. Since lists are bad for parallelism, so in this homework we will investigate similar higher-order functions on trees.

Previously, we used trees that had data at each internal node. For this week, we will instead consider trees where there is data only at the leaves:

An `'a tree` is either

1. empty
2. a leaf with value `x:'a`
3. a node with two subtrees

and that's it!

This is represented by the following datatype:

```
datatype 'a tree = Empty | Leaf of 'a | Node of 'a tree * 'a tree
```

This datatype is defined in `lib.sml`, which is loaded by your homework file. `lib.sml` also provides

```
val fromlist : 'a list -> 'a tree
val tolist : 'a tree -> 'a list
```

which you can use for testing (but only for testing!). Since we are mostly concerned with the contents of the trees, and not their specific arrangement, we can write tests that convert lists to trees and back, which has a more concise notation than writing the trees themselves. There are some examples in the homework code, which you can add more tests to.

We have also provided a function `sort`

```
sort : ('a * 'a -> order) * 'a tree -> 'a tree
```

based on the tree sorting code from earlier in the semester (together with some additional code to convert the leaf-empty-node trees used here to the leaf-node trees used in sorting, and vice versa).

`sort` takes a comparison function, which produces an `order`, which is datatype whose constructors are `LESS` and `EQUAL` and `GREATER`:

```
datatype order = LESS | EQUAL | GREATER
```

It sorts the tree into increasing order according to this comparison function. There are built-in comparison functions on integers (`Int.compare`) and floating points (`Real.compare`). For example,

```
Real.compare(3.0,2.0) == GREATER
Real.compare(2.0,3.0) == LESS
Real.compare(2.0,2.0) == EQUAL


sort (Real.compare , (Node(Node(Leaf 3.0, Leaf 2.0),
                          Node(Leaf 4.0, Leaf 1.0))))
==  Node (Node (Leaf 1.0,Leaf 2.0),
          Node (Leaf 3.0,Leaf 4.0))
```

1 Map

Consider the following two functions:

```
(* pluralize_rec t evaluates to a tree t', where
  t' has the same structure as t, and the string at each leaf of t'
  is the string at the corresponding position in t, with an 's'
  affixed to the end. *)
fun pluralize_rec (t : string tree) : string tree =
    case t of
        Empty => Empty
      | Leaf x => Leaf (x ^ "s")
      | Node(l,r) => Node(pluralize_rec l , pluralize_rec r)

(* mult_rec t evaluates to a tree t', where
   t' has the same structure as t, and the int at each leaf of t'
   is the int at the corresponding position in t, multiplied by c
*)
fun mult_rec (c : int, t : int tree) : int tree =
    case t of
        Empty => Empty
      | Leaf x => Leaf (c * x)
      | Node(l,r) => Node(mult_rec (c,l) , mult_rec (c,r))
```

The pattern is "compute a new tree by applying some function `f` to each element of the given tree."

Task 1.1 (8 pts). Write a higher-order function

```
map : ('a -> 'b) * 'a tree -> 'b tree
```

that abstracts this pattern.

Task 1.2 (2 pts). Rewrite `pluralize` and `mult` using `map`. The functions should be equivalent to `pluralize_rec` and `mult_rec`, but should be defined using `map` rather than being defined directly by recursion.

2 Reduce

Consider the following two functions:

```
(* sum_rec t evaluates to a number n, where n is
   the sum of all of the numbers at the leaves of t *)
fun sum_rec (t : int tree) : int =
    case t of
        Empty => 0
      | Leaf x => x
      | Node(t1,t2) => (sum_rec t1) + (sum_rec t2)

(* join_rec t evaluates to a string s, where s is
   the concatenation of all of the strings at the leaves of t,
   in order from left to right *)
fun join_rec (t : string tree) : string =
    case t of
        Empty => ""
      | Leaf x => x
      | Node(t1,t2) => (join_rec t1) ^ (join_rec t2)

val "abc" = join_rec(Node(Leaf "a", Node(Leaf "b", Leaf "c")))
```

The general pattern here is called `reduce`, which takes a binary function of type `'a * 'a -> 'a` to apply at each node, and a value of type `'a` for the empty tree, and computes an `'a` from an `'a tree`.

Task 2.1 (8 pts). Write the function

```
reduce : ('a * 'a -> 'a) * 'a * 'a tree -> 'a
```

that implements the operation of reduction on trees.

Task 2.2 (2 pts). Rewrite `sum` and `join` using `reduce`. The functions should be equivalent to `sum_rec` and `join_rec`, but should be defined using `reduce` rather than being defined directly by recursion.

3 Programming with map and reduce

To receive credit for a task in this section, your function must not be defined recursively. The goal is to practice programming by combining higher-order functions. In each task, you may use `map`, `reduce`, any previous tasks in this section, and any other functions that are specifically allowed by the task. If you cannot figure out how to solve a task this way, you may wish to first define it recursively, and then think about how the recursive version can be rewritten with higher-order functions.

We say that `x` is an *element* of a tree `t` if `Leaf x` occurs somewhere in `t`. In all of these tasks, the shape of the resulting tree is up to you, as long as it has the correct elements in the order specified in the problem. For example, the following trees have all of the same elements in the same order:

```
Node(t1,Node(t2,t3))  and  Node(Node(t1,t2),t3)
Node(Empty,t)   and  t
Node(t,Empty)   and  t
```

so one side is always just as correct an answer as the other. In particular, you never need to rebalance a tree.

3.1 Flatten

The type `('a tree) tree` represents a tree whose leaves are themselves trees.

Task 3.1 (5 pts). Write a function

```
flatten : ('a tree) tree -> 'a tree
```

such that `flatten t` contains all of the elements of all of the trees in `t`. The elements of each tree `t1` in `t` should occur in `flatten t` in the same order in which they occur in `t1`; if a tree `t1` is to the left of a tree `t2` in `t`, the elements of `t1` should occur to the left of the elements of `t2` in `flatten t`.

For example:

```
   flatten (Node (Leaf (Node (Leaf 1, Leaf 2)),
                  Node (Leaf (Leaf 3),
                        Empty)))
== Node (Node (Leaf 1,Leaf 2),Node (Leaf 3,Empty))
```

3.2 Filter

Task 3.2 (5 pts). Define a function

```
filter : ('a -> bool) * 'a tree -> 'a tree
```

such that `filter (p, t)` contains all and only the elements `x : 'a` of `t` for which `p x` returns `true`. The elements that are kept should be in the same order as in the original tree.

For example:

```
    filter (fn x => x > 2, Node (Node (Leaf 1,Leaf 2),Node (Leaf 3,Empty)))
 == Node (Node (Empty,Empty),Node (Leaf 3,Empty))
```

Hint: first create an `('a tree) tree` where each `Leaf x` in the original tree is replaced by `Leaf (Leaf x)` if `x` is to be kept, or `Empty` if it is not.

3.3 All Pairs

Task 3.3 (5 pts). Define a function

```
    allpairs : 'a tree * 'b tree -> ('a * 'b) tree
```

such that `allpairs(t1, t2)` contains the pair `(x,y)` for every element `x` of `t1` and `y` of `t2`. The order of the pairs is unspecified.

For example,

```
     allpairs (Node(Leaf 1, Leaf 2), Node(Leaf "a", Leaf "b"));
 ==  Node (Node (Leaf (1,"a"),Leaf (1,"b")),Node (Leaf (2,"a"),Leaf (2,"b")))
```

3.4 Partnr

You are writing an app to help students find study partners. Each student fills out a survey, which, for simplicity, we will assume consists of the four questions in Figure 1. That is, each person provides a tuple

```
 (username,answer1,answer2,answer3,answer4) : string * int * int * int * int
```

where `username` is a string identifying the person, and each `answerN` is the number of that person's answer to question N.

For convenience, we abbreviate the tuple of `int`'s by the type abbreviation

```
    type answers = int * int * int * int
```

For example,

```
("drl", 5, 2, 1, 2) : string * answers
```

means that my answers are "all of the above", "in the computer lab", "music", and "let's do it live."

Scoring Functions A *scoring function* is a function that takes two people's answers and computes a score, which is a floating point number, where higher numbers indicate higher study partner compatibility.

For example, here is a simple scoring function, which totals up the number of answers that two people have in common:

```
fun same(x : int, y : int) : real =
    case x = y of
        true => 1.0
      | false => 0.0
fun count_same ((a1,a2,a3,a4) : answers , (a1',a2',a3',a4') : answers) : real =
    same (a1,a1') + same (a2,a2') + same (a3,a3') + same (a4,a4')
```

Scoring functions should be symmetric: `score(a1,a2) == score(a2,a1)`.

Task 3.4 (2 pts). Write another scoring function `my_scoring`, which implements some alternative compatibility scoring of your choice. Explain in a comment why you think your scoring mechanism would provide good results. For example, other scoring functions might give different weights to different questions, or allow "fuzzy matching" of answers (e.g. if one person likes to study in the morning and another likes to study any time, maybe they are somewhat compatible.)

Matching Your goal is to analyze the given data and report a ranked list of possible study partners. You are given a scoring function and a cutoff, and should only report possible partners whose scores are above the cutoff.

Task 3.5 (13 pts). Write a function

```
fun matches (similarity : answers * answers -> real,
             cutoff : real,
             people : (string * answers) tree)
            : (string * string * real) tree = ...
```

where `similarity` is a scoring function, `cutoff` is a real number, `people` is the input data for all of the users. `matches (similarity, cutoff, people)` should compute a tree of pairs `(person1,person2,score)` where

- each `score` is the similarity score of `person1` and `person2`
- the tree is sorted from highest scores to lowest scores
- only pairs of people whose score is bigger than `cutoff` are included
- the tree never contains a pair of people of the form `(person1,person1,_)` or both the pair `(person1,person2,_)` and the pair `(person2,person1,_)`.

Hints:

- Start by making the tree of all pairs of people.

1. What time do you like to do your homework?

   1 Morning

   2 Afternoon

   3 Evening

   4 Late-night

   5 All of the above

2. What's your preferred brand of skype?

   1 Hangouts

   2 Zoom

   3 Facetime

3. What kind of background noise helps you study?

   1 Music

   2 TV

   3 Quiet, please!

4. How do you like to spread your work for a particular class out over the week?

   1 Slow and steady wins the race

   2 Let's do it live

Figure 1: Survey

- Use `<` to test whether one string is less than another, according to lexicographic order.
- Use `sort` (described at the beginning of the handout) to sort trees.
- You can use the function `show_matches` to print the results in a nice way. For example, we have provided

  ```
      val test_data : (string * answers) tree =
          fromlist [ ("A",(1,1,1,1)), ("B",(2,2,2,2)), ("C",(1,2,2,2)) ]
  ```

  and on this input, we have

  ```
  - show_matches (count_same, 0.0, test_data);
  B and C have compatibility 3.0
  A and C have compatibility 1.0
  ```

4 NON-COLLABORATIVE CHALLENGE PROBLEM: Patterns

Remember that non-collaborative challenge problems are to be done independently. You are not allowed to communicate with anyone about the problems, except to ask the instructor or TAs clarification questions (not hints). Additionally, you are not allowed to search for help on the specific problem from any sources besides the course materials.

In this problem, you will prove a specification about the `merge` function on trees from Lecture 11/12, and the `depth` function on trees from Lab 5. To avoid confusion, remember that these examples used a different tree type than used above (in Homework 7):

```
datatype tree = Empty | Node of tree * int * tree
```

So for this problem there is no `Leaf` constructor and `Node`'s have a left subtree, a number at the root, and a right subtree.

The code for these funtions is

```
fun depth (t : tree) : int =
    case t
     of Empty => 0
      | Node(l,_,r) => 1 + Int.max(depth l, depth r)

fun splitAt (t : tree , bound : int) : tree * tree =
 case t of
     Empty => (Empty , Empty)
   | Node (l , x , r) =>
      (case bound < x of
           true => let val (ll , lr) = splitAt (l , bound)
                   in (ll , Node (lr , x , r))
                   end
         | false => let val (rl , rr) = splitAt (r , bound)
                    in (Node (l , x , rl) , rr)
                    end)

fun merge (t1 : tree , t2 : tree) : tree =
 case t1 of
     Empty => t2
   | Node (l1 , x , r1) =>
      let val (l2 , r2) = splitAt (t2 , x)
      in
          Node (merge (l1 , l2) ,
                x,
                merge (r1 , r2))
      end
```

For the running time analysis of `mergesort` on trees, we needed the following property:

Theorem 1. *For all trees* `t1:tree` *and* `t2:tree`*,*

$$\texttt{depth(merge(t1,t2))} \leq \texttt{depth(t1)} + \texttt{depth(t2)}$$

Task 4.1 (20 pts). Prove this theorem. State and prove any lemmas about `splitAt` that you need. You can use the properties of `max` (maximum) from Homework 5 without proving them.

Because the problem is a little open-ended, there is no templated written handin for this part, but you can start with the template for induction on trees from Homework 5, and the example in the Lab 5 solutions.