ML Assignment

Due Friday, November 20

Your assignment is to write in ML, using the SML/NJ system, a series of simple definitions (types and functions) for computing on lists and trees. Obviously, the code should be purely functional. Also, you must not look online for the solutions (nor get the code from any other source).

You should make use of ML's pattern matching facility for defining functions and extracting elements from lists and tuples. For example, do not use the `hd` and `tl` functions to extract the head and tail of a list. Also, if you need to check if a list contains a single element, use a pattern (e.g.,`[x]`) rather than the `length` function.

Finally, be sure to read the section at the bottom of this assignment labeled "Important Notes".

1. In preparation for writing a partition sort function (below), define a function `partition` that takes an integer `x` and an integer list `L` as parameters and partitions `L` based on `x`. That is, `partition` returns a tuple of two lists, such that the first list contains all elements of `L` less than `x` and the second list contains the elements of `L` that are greater than or equal to `x`.

   An example of the use of `partition` (in the SML/NJ system) would be:

   ```
   - partition 6 [5,2,8,4,1,9,6,10];
   val it = ([5,2,4,1],[8,9,6,10]) : int list * int list
   ```

2. Implement a partition sorting function, `partitionSort` which uses your `partition` function, above, to sort a list `L` of integers, returning a sorted list of the elements of `L`. A partition sort is essentially Quicksort, but without the property of using only a constant amount of space. The algorithm, given a list `L`, is:

   - If `L` is empty or contains a single element, return `L` (use pattern-matching for these two cases, not a conditional).
   - Otherwise, assuming `L` is of the form `(x::xs)`, partition `xs` into two lists based on `x` by calling your `partition` function.
   - Recursively call `partitionSort` on each of the two lists, to sort each of the lists.
   - Return a single sorted list containing the elements from the first sorted list, followed by `x`, followed by the elements of the second sorted list.

   A use of `partitionSort` would be:

   ```
   - partitionSort [5,2,9,10,12,4,8,1,19];
   val it = [1,2,4,5,8,9,10,12,19] : int list
   ```

3. Implement a polymorphic partition sort function, `Sort`, that will sort a list of elements of any type. In order to do this, though, `Sort` must take an additional parameter, the `<` (less-than) operator, that operates on the element type of the list. For example, two uses of `Sort` would be:

   ```
   (* Using the built-in < for comparing integers. The compiler is
      smart enough to figure out which < to use *)
   - Sort (op <) [1,9, 3, 6, 7];
   val it = [1,3,6,7,9] : int list
   ```

```
(* sorting a list of lists, where the less-than operator compares
   the length of two lists *)
- Sort (fn(a,b) => length a < length b) [[1, 9, 3, 6], [1], [2,4,6], [5,5]];
val it = [[1],[5,5],[2,4,6],[1,9,3,6]] : int list list
```

Note that you will also need to create a helper function for partitioning the list (analogous to the `partition` function you wrote above). This time, though, nest the helper function inside the `Sort` function using `let`.

4. Define a polymorphic binary tree datatype (i.e. `datatype 'a tree = ...`) such that a tree may be:

   - the empty tree,
   - a leaf labeled with an `'a`, or
   - an interior node which is labeled with an `'a` and has two children that are both of type `'a tree`.

   For example, your datatype declaration should allow the following tree to be constructed:

```
val myTree = node(5,node(2,leaf 3,empty), node(12,node(8,leaf 7,leaf 11),
                    node(4,leaf 1, node(6,empty, leaf 9))))
```

5. Define the polymorphic `maxTree` function that, given an `'a tree`, for any type `'a`, finds the maximum label value in the tree. `maxTree` should:

   - take the `<` operator as a parameter, just like in your `Sort` code, and
   - since `empty` doesn't have a label, raise an exception if `maxTree` is ever called on the empty tree (see below for raising an exception in ML).

   For example, given the definition of `myTree` above,

```
- maxTree (op <) myTree;
val it = 12 : int
```

   and

```
- (* passing in > as the "less than" operator, should give the minimum! *)
maxTree (op >) myTree;
val it = 1 : int
```

6. Define the function `Labels` that, given a tree, returns a list of the labels in the tree. However, `Labels` must take as an additional parameter a function that determines the order of the labels in the resulting list. The orders that should be supported are pre-order, in-order, and post-order (see the Wikipedia page on "Tree Traversal"). You should also define the functions `preorder`, `inorder`, and `postorder` that can be used in calls to `Labels` as follows.

```
- fun preorder ... = ...  (* fill this in *)
...
- fun inorder ... = ...  (* fill this in *)
...
- fun postorder ... = ...  (* fill this in *)
...
```

```
- (* calling Labels using the above functions and myTree, above *)
Labels preorder myTree;
val it = [5,2,3,12,8,7,11,4,1,6,9] : int list
- Labels inorder myTree;
val it = [3,2,5,7,8,11,12,1,4,6,9] : int list
- Labels postorder myTree;
val it = [3,2,7,11,8,1,9,6,4,12,5] : int list
```

7. Define a polymorphic function, `lexLess`, that performs a lexicographic "less than" comparison on two lists. That is, a list L_1 is lexicographically less than a list L_2 iff:

   - there is a k such that the first $k-1$ elements of L_1 and L_2 are equal and the k^{th} element of L_1 is less than the k^{th} element of L_2, or
   - L_1 is of length k_1 and L_2 is of length k_2, such that $k_1 < k_2$ and the first k_1 elements of the two lists are equal.

   Note that lexicographic ordering is used to order words in a dictionary.

   Since the corresponding elements of the two lists will have to be compared, a `<` operator for the element type will have to be passed as a parameter to `lexLess`. For example,

```
- lexLess (op <) [1,2,3,4] [1,2,3,4,5];  (* uses integer < *)
val it = true : bool
- lexLess (op <) [1,2,3,4] [1,2,3];
val it = false : bool
- lexLess (fn(L1,L2) => length L1 < length L2)
          [[3,4,5,6],[1,2,4]] [[1,2,3,4],[1,2,3,4]];
val it = true : bool
```

   You should avoid using the built-in equality (`=`) operator in your definition of `lexLess`, otherwise you are likely to get the wrong answer for the last example, above.

8. Define the polymorphic function `sortTreeList` which sorts an `'a tree list` for any type `'a`. Given two `'a tree`s, `T1` and `T2`, the tree `T1` is less than `T2` if the list of `T1`'s labels in in-order traversal order is lexicographically less than the list of `T2`'s labels in in-order order. Of course, `sortTreeList` will need to take as a parameter the `<` operator for comparing labels in the trees.

   For example, given the following tree definitions,

```
(* Labels are [1,2,3,4,5,6,7] *)
val t1 = node(5,node(4,node(2,leaf 1, leaf 3), empty),
                node (7,leaf 6, empty))

(* Labels are [0,1,2,3,4,5,6,7] *)
val t2 = node (5,node(4,node (2,node(1,leaf 0, empty),leaf 3),empty),
                 node (7,leaf 6,empty))

(* Labels are [1,2,3,4,5,6] *)
val t3 = node(5,node(4,node(2,leaf 1, leaf 3), empty), leaf 6)
```

   here is the result of calling `sortTreeList` on a list of the above three trees.

```
- sortTreeList (op <) [t1, t2, t3];
val it =
  [node
     (5,node (4,node (2,node (1,leaf 0,empty),leaf 3),empty),
      node (7,leaf 6,empty)),
   node (5,node (4,node (2,leaf 1,leaf 3),empty),leaf 6),
   node (5,node (4,node (2,leaf 1,leaf 3),empty),node (7,leaf 6,empty))]
  : int tree list
```

In your definition of `sortTreeList`, be sure to use the functions you previously wrote.

Important Notes

- You should put your code in a file with a `.sml` extension. To load a file containing ML code into the SML/NJ system, type

  ```
  use "filename.sml";
  ```

  When you are finished with the assignment, upload just the file containing your definitions. Be sure to use exactly the same function and type names as specified above.

- Put the following lines at the top of your file, to tell the SML/NJ system the maximum depth of a datatype to print and the maximum length of a list to print.

  ```
  Control.Print.printDepth := 100;
  Control.Print.printLength := 100;
  ```

  If you don't put these lines in your file, the system will only print a limited number of elements of a list, or to a limited depth in a datatype (such as a tree), after which it prints `#` to save space.

- To define an exception in ML (in its simplest form), you write
  `exception` *exception_name*
  To raise the exception, you write
  `raise` *exception_name*
  For example,

  ```
  exception DivByZero

  fun myMod x 1 = raise DivByZero
   |  myMod x y = x mod (y-1)
  ```

  See the ML online references for defining exceptions that take parameters and for handling exceptions (neither of which you need to do for this assignment).

- To declare a function that takes a function as a parameter, such that the parameter is used as an infix operator, use the `(op ...)` syntax as shown below.

  ```
  - fun f (op <) x y = if x < y then "less" else "not less";
  val f = fn : ('a * 'b -> bool) -> 'a -> 'b -> string

  - (* passing a lambda expression as the < operator *)
  f (fn(a,b) => (a*a) < (b*b)) ~4 3;
  val it = "not less" : string
  ```

Note that negative numbers are written using `~`, not `-`.

- Be mindful of the difference between a function that takes two parameters, e.g.

  ```
  fun f x y = x + y
  ```

  and a function that takes a single parameter that is a tuple with two elements, e.g.

  ```
  fun g (x,y) = x + y
  ```

  Any function used as an infix operator, e.g. passed as a parameter to a function such as

  ```
  fun h (op <) x y = x < y
  ```

  must take a tuple as a parameter.

- Do not copy-and-paste the sample code from this PDF document into your program or into the SML system, since some symbols may be stored differently in PDF than in plain text. I have provided a plain text version of the sample code along with this PDF.