**ML Assignment**

**Due Tuesday, November 18**

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.

1. Implement a partition sort function, `intPartitionSort`, for a list of integers. A partition sort (similar to Quicksort, but not in-place), given a list `(x::xs)`, should partition the elements of `xs` into two sublists: one list with numbers less than `x` and the other list with numbers not less than `x`. Then, the partition sort is recursively called on each of the two sublists and the final result is constructed from appending the resulting lists, along with the list containing x, in the appropriate order. You can write it all as one function or break it up into several functions.

   A use of `intPartitionSort` (in the SML/NJ system) would be:

   ```
   - intPartitionSort [3,5,1,8,4];
   val it = [1,3,4,5,8] : int list
   ```

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

   ```
   (* Using the built-in < for comparing integers. The compiler is
      smart enough to figure out which < to use *)
   - partitionSort (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 *)
   - partitionSort (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
   ```

3. Define a polymorphic tree datatype (i.e. `datatype 'a tree = ...`) such that a leaf is labeled with an `'a` and an interior node has a list of children, each of type `'a tree`. That is, each interior node can have an arbitrary number of children, rather than just two (as in a binary tree). For example, your datatype declaration should allow the following tree to be constructed:

   ```
   val myTree = node [node [node [leaf [4,2,14],leaf [9,83,32],leaf [96,123,4]],
                            node [leaf [47,71,82]],node [leaf [19,27,10],
                                                         leaf [111,77,22,66]],
                            leaf [120,42,16]],
                      leaf [83,13]]
   ```

4. Define a polymorphic `sortTree` function that, given an `'b list tree` (for some type `'b`, so that each leaf has a list of elements of type `'b`) returns a new tree that is identical to the original tree, except that the list at each leaf is sorted. It must use the polymorphic partition sort function that you wrote (above), so therefore must also take the `<` (less-than) operator as a parameter. A use of `sortTree` is:

```
(* Here the parameter is an int list tree *)
- sortTree (op <) (node [leaf [4,2,3,1], leaf [7,2,5,0]]);
val it = node [leaf [1,2,3,4],leaf [0,2,5,7]] : int list tree
```

5. Define a polymorphic `merge` function that takes two sorted lists and returns a sorted list containing the elements of both lists. Since it is polymorphic, it also needs to take the `<` operator as a parameter. A use of `merge` would be:

   ```
   (* Sorting in decreasing order *)
   - merge (fn (a,b) => a > b) [8,6,4,2] [7,5,3,1];
   val it = [8,7,6,5,4,3,2,1] : int list
   ```

6. Finally, define a polymorphic function `mergeTree` that, given an `'c list tree` (for some type `'c`) returns a sorted list (of type `'c list`) of all the elements found at all the leaves of the tree. It should build on code that you wrote for the previous parts of this assignment and use both the `sortTree` and `merge` functions. For example, given the tree m̂yTree that I defined above,

   ```
   (* Again, myTree is an int list tree, so sortTree will return an int list *)
   mergeTree (op <) myTree;
   val it =
     [2,4,4,9,10,13,14,16,19,22,27,32,42,47,66,71,77,82,83,83,96,111,120,123]
     : int list
   ```

**Hints/Suggestions**

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

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

  When you are finished with the assignment, submit just the file containing your definitions. Be sure to use 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.

- If you want a function to return several values, you can have the function return a tuple. This works well with tuple pattern-matching, as seen in the following:

  ```
  let
    fun f x y = (x+1, y-1)
    val (a,b) = f 3 4
  in
    a+b
  end
  ```

- In ML, the behavior of infix operators can be user defined, as follows:

```
(* Tells the compiler that == will be used as an infix operator *)

infix ==

(* Defines an infix function named ==.  Note that the type of ==, in this
   case, will be:  'a list * 'a list -> bool *)

fun [] == [] = true
  |  (x::xs) == (y::ys) = x = y andalso xs == ys
  |  _ == _ = false;



(* This takes a function as its first parameter, where the formal
   parameter is named "==" and is infix. *)

fun foo (op ==) L1 L2 =
   if L1 == L2 then L1 else L2


(* can pass a user-defined function as the first argument. Infix operators
   always have to take a two-element tuple as a parameter.  In this case,
   because of how foo is defined, it would have to return a bool. *)

val result1 = foo (fn (a,b) => length a = length b) [3, 4, 5] [6,7,8]


(* can pass an existing operator as the first argument *)
val result2 = foo (op =) 3 4
```

- Consider using the built-in `map` function in your code. It works just like `map` in Scheme, e.g.

```
- map (fn x => x+1) [3,4,5];
val it = [4,5,6] : int list
```

- You might also want to use the built-in `foldr` function, which is ML's version of the reduce operator. It's already built-in, but is easily defined by

```
fun foldr _ b [] = b
 |  foldr f b (x::xs) = f(x, foldr f b xs)
```

so that

```
- foldr (op +) 0 [1,2,3];
val it = 6 : int
```