Programming Languages
Spring 2017

ML Assignment
Due Sunday, April 23

Your assignment is to write in ML, using the SML/NJ system, a series of simple function and type definitions. Obviously, the code should be purely functional.

Important: Read the "Hints and Suggestions" section at the bottom of this assignment.

1. Define a polymorphic linear-time procedure, `reverse L`, where `L` is of type `'a list`, that returns a list containing the same elements as `L`, but in reverse order. You can use the same algorithm as shown in the Scheme lecture. You can also define your own helper function if you like (as I did in Scheme). Remember that to be linear time, `reverse` should not use the append operator, `@`.

   For example,

   ```
   reverse [1,2,3,4,5];
   val it = [5,4,3,2,1] : int list
   - reverse [[1,2],[3,4],[5,6]];
   val it = [[5,6],[3,4],[1,2]] : int list list
   ```

   Note that `reverse` does not reverse any nested lists within `L`. Your code for this part should be around 3 lines in total.

2. If you defined a separate helper function outside of reverse, write a function `new_reverse L`, where `L` is of type `'a list`, that also reverses `L` but does not require a helper function outside of `new_reverse`. Note that you can nest a helper function inside of `new_reverse` using a `let`. Your code should be no more than 5 lines.

   If you did not need a helper function outside of your `reverse` function, above, then you can skip this part.

3. Write a polymorphic function, `reduce_depth L`, where `L` is of type `'a list list` (i.e. each element of the list `L` is an `'a list`), that returns a list of type `'a list` whose elements are the elements of the elements of `L`. That is, the result is a list whose depth of nesting is one less than `L's depth`. For example,

   ```
   - reduce_depth [[1,2],[3,4],[5,6]] ;
   val it = [1,2,3,4,5,6] : int list

   - reduce_depth [[[1,2],[3,4]],[[5,6],[7,8]]] ;
   val it = [[1,2],[3,4],[5,6],[7,8]] : int list list
   ```

   Your code should be roughly 2 lines.

4. In ML, a set of integers can be representing as a list of integers, as long as that list has no duplicate elements and is interpreted as being unordered. For this problem, define the following infix operators:

   - `elt` (membership test), such that `2 elt [1,2,3]` returns true and `2 elt [1,3,4]` returns false.

- `++` (set union), such that `[3,1,2,4] ++ [4,5,6,3]` returns `[1,2,4,5,6,3]` (in any order).
- `**` (set intersection), such that `[3,1,2,4] ** [4,5,6,3]` returns `[3,4]` (in any order).
- `--` (set difference), such that `[3,1,2,4] -- [4,5,6,3]` returns `[1,2]` (in any order).

For example,

```
- [1,2,3,4]++([3,4,5,6,7,8] ** ([9,8,7,10] -- [10,11,12])) ;
val it = [1,2,3,4,7,8] : int list
```

Remember to use an `infix` declaration for each of these operators, in addition to the definition of the operator itself. The definition of each operator should be rougly two lines (in addition to the one-line `infix` declaration).

5. Define a tree datatype, `tree`, where a `leaf` has an integer label and an interior `node` has an integer label and two child trees. For example,

```
- node (5, leaf 6, leaf 7);
val it = node (5,leaf 6,leaf 7) : tree
```

This definition should be one line of code.

6. Define a polymorphic version of your tree type, `ptree`, such that for any type `'a` , a leaf has a label of type `'a` and an interior node has a label of type `'a` and two ptrees as children. To avoid a naming conflict with your code from part 5, above, use `pleaf` and `pnode` to identify a leaf and a node of your `ptree` type, respectively.

For example,

```
- pnode (5, pleaf 6, pleaf 7);
val it = pnode (5,pleaf 6,pleaf 7) : int ptree

(* defining a ptree variable *)
- val myptree = pnode ([1,2], pnode ([3,4], pleaf [5,6], pleaf [7,8]),
                      pnode ([9,10], pleaf [11,12], pleaf [13,14])) ;
val myptree =
  pnode
    ([1,2],pnode ([3,4],pleaf [5,6],pleaf [7,8]),
     pnode ([9,10],pleaf [11,12],pleaf [13,14])) : int list ptree
```

The `ptree` definition should be one line of code.

7. Write a function `interior T`, where `T` is an `'a ptree`, that returns a list of the labels at the *interior* nodes of `T`. The order of the list should reflect an in-order traversal of the tree.

For example,

```
- interior myptree;   (* myptree is defined above *)
val it = [[3,4],[1,2],[9,10]] : int list list
```

Your code should be roughly two lines.

8. Define the function `mapTree f T`, where `f` is of type `'a -> 'b` and `T` is an `'a ptree`, that returns a new tree of type `'b ptree`. The new ptree should have the same structure as T, but differ only in its labels. In particular, each label at a pleaf or pnode of the new ptree results from applying `f` to the label of the corresponding pleaf or pnode of `T`, respectively.

   For example,

```
- mapTree (fn a => length a) myptree;   (* myptree is defined above *)
val it = pnode (2,pnode (2,pleaf 2,pleaf 2),pnode (2,pleaf 2,pleaf 2))
  : int ptree
```

9. Define a function, `lexLess (op <) L1 L2`, where `L1` and `L2` are both of type `'a list` and `<` is of type `'a * 'a -> bool`, that performs a lexicographic *less than* comparison using the passed-in `<` operator. A lexicographic *less than* operation on lists is defined as follows:

   - `[]` is less than any non-empty list.
   - $(x::xs)$ is less than $(y::ys)$ iff:
     - $x < y$, or
     - $x = y$ and xs is lexicographically less than ys.

   Note that the test for equality in your code should not use the built-in `=` operator, because it may not have the behavior you want. For this problem, $x = y$ is true iff $(x < y)$ is false and $(y < x)$ is false.

   For example,

```
- lexLess (fn (L1,L2) => length L1 < length L2) [[1,2],[3,4]] [[5],[6,7,8]] ;
val it = false : bool
- lexLess (fn (L1,L2) => length L1 < length L2) [[5],[6,7,8]] [[1,2],[3,4]] ;
val it = true : bool
- lexLess (fn (L1,L2) => length L1 < length L2) [[5,6],[7,8]] [[1,2],[3,4,5]] ;
val it = true : bool
```

   This function can be written in roughly 4 lines.

10. Define a function `ptreeLess (op <) T1 T2`, where `T1` and `T2` are of type `'a ptree` and `<` is of type `'a * 'a -> bool`, that returns true iff `T1` is less than `T2`. For this assignment, the *less than* operator on `ptree`'s is defined as follows:

    - a pleaf is less than a pnode.
    - a pleaf with a label x is less than a pleaf with a label y iff x is less than y.
    - a pnode with a label x and subtrees $l1$ and $r1$ is less than a pnode with label y and subtrees $l2$ and $r2$ iff:
      - $l1$ is less than $l2$, or
      - $l1 = l2$ and x is less than y, or
      - $l1 = l2$ and $x = y$ and $r1$ is less than $r2$

    For example,

```
- ptreeLess (op <) (pleaf 3) (pleaf 4); (* compiler infers that < is the integer comparison
val it = true : bool
- ptreeLess (op <) (pleaf 4) (pleaf 3);
val it = false : bool
- ptreeLess (op <) (pleaf 5) (pnode (4, pleaf 3, pleaf 2)) ;
val it = true : bool
- ptreeLess (op <) (pnode (4,pleaf 5, pleaf 6)) (pnode (6, pleaf 5, pleaf 2));
val it = true : bool
- ptreeLess (op <) (pnode (6,pleaf 5, pleaf 6)) (pnode (6, pleaf 5, pleaf 2));
val it = false : bool
- ptreeLess (op <) (pnode (6,pleaf 5, pleaf 6)) (pnode (6, pleaf 5, pleaf 7));
val it = true : bool
- ptreeLess (op <) (pnode (6,pleaf 5, pleaf 6)) (pnode (7, pleaf 6, pleaf 2));
val it = true : bool
```

As in the previous problem, the built-in = operator should not be used to test equality. Note that the `<` operator passed in to `ptreeLess` should only be used for comparing labels. The code for `ptreeLess` should be 7 lines or so.

Important: The `ptreeLess` function should be as efficient as possible, so it should never call the same function or operator twice with exactly the same arguments. For example, `x < y` should only appear once in the body of the function, since `<` can be an arbitrarily complex function.

Hints/Suggestions

- 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, submit 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. Important: The two lines above should end with the semicolons that you see. However, semicolons should not appear anywhere else in your file.

- 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
```