Name: ***Solutions***

PART I, the night before

1) [8 pts] Implement a recursive function `common_prefix` which takes two lists and returns the longest list that is a prefix of both argument lists. It should use the built-in `=` function to compare elements. The following are some example calls:

```
common_prefix([3, 4, 5], [3, 4, 6, 7]) → [3, 4]
common_prefix([3, 4, 5], [3, 4]) → [3, 4]
common_prefix([5, 3, 4], [3, 4, 5, 6, 7]) → []
```

```
fun common_prefix(x::xs, y::ys) =
     if x=y then x::common_prefix(xs, ys) else []
  | common_prefix(_, _) = []
```

2) a) [8 pts] Show how to implement `filter` using `reduce`. (Use the versions of these functions done in class.)

```
fun filter(pred, lst) =
  reduce(fn(elem,rest)=>
            if pred(elem) then elem::rest else rest,
         [], lst)
```

b) [4 pts] Why are lexically scoped nested functions (which C lacks) critical in order to use `reduce` to implement `filter`?

Otherwise, it would be very difficult for the argument function to reduce to refer to pred in its body, since pred is a local variable of the lexically enclosing scope.

3) Consider the following polymorphic binary tree datatype declaration:

```
datatype 'a BTree =
    Empty
  | Node of {left:'a BTree, value:'a, right:'a BTree}
```

Consider a higher-order function `reduce_infix` that takes a function, a base value, and a binary tree, and visits all the nodes of the tree **in left-to-right infix order**, calling the argument function on each element value stored at the nodes, starting from the given base value. The following is an example call of `reduce_infix` on an example binary tree:

```
val t1 = Node{left=Node{left=Empty,value=”a”,right=Empty},
              value=”b”,
              right=Node{left=Empty,value=”c”,right=Empty}}
```

```
val t2 = Node{left=Empty,
              value=”e”,
              right=Node{left=Empty,value=”f”,right=Empty}}
val t3 = Node{left=t1,value=”d”,right=t2}
(* concatenate all the strings in the tree *)
reduce_infix(fn(elem,prevs)=> prevs ^ elem, ””, t3)
(* evaluates to ”abcdef” *)
```

a) [4 pts] What is the most general type of the `reduce_infix` function?

```
('a * 'b -> 'b) * 'b * 'a BTree -> 'b
```

b) [10 pts] Implement `reduce_infix`.

```
fun reduce_infix(f, b, Empty) = b
  | reduce_infix(f, b, Node{left,value,right}) =
       let val l = reduce_infix(f, b, left)
           val m = f(value, l)
           val r = reduce_infix(f, m, right)
       in r end
```

c) [7 pts] Use `reduce_infix` to define a function `toList` that constructs a list of all the elements in the tree, in left-to-right infix order. (Note that this is not the most efficient way to do this; a right-to-left reduction would be more suitable.)

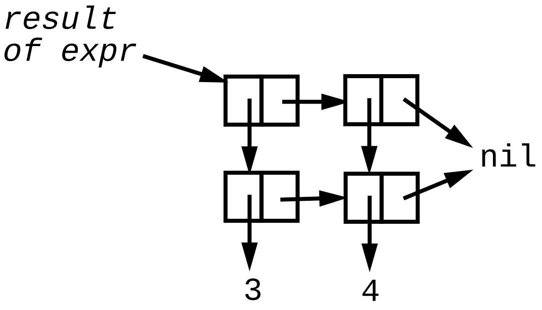
```
fun toList(t) =
  reduce_infix(fn(elem,prevs)=> prevs @ [elem], [], t)
```

d) [3 pts] Why is it hard to write a tail-recursive version of `reduce_infix`?

Because there are two recursive calls, and they can’t both be last.

PART II, in class

4) a) [4 pts] For the following box-and-pointer diagram (where the boxes in the diagram are all list cons cells), give a **single** ML expression that evaluates to the data structure, with the same sharing relationships. Hint: exploit `let`.



```
let val shared = [4] in [3::shared, shared] end
```

b) [2 pt] For the above diagram, show how SML would print out the data structure.

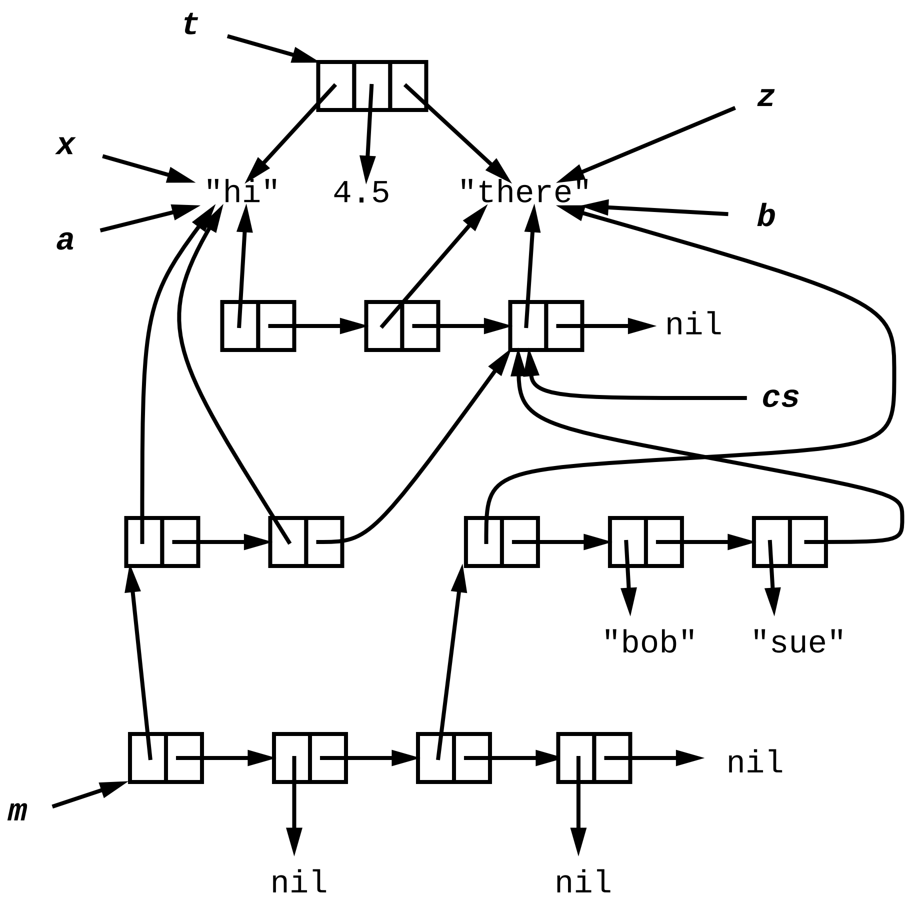
```
[[3, 4], [4]]
```

c) [2 pt] What is the type of this data structure?

```
int list list
```

d) [8 pts] For the following sequence of ML expressions, illustrate using a box-and-pointer diagram the final data structures resulting from evaluating the sequence, showing where the variables `t`, `x`, `z`, `a`, `b`, `cs`, and `m` point into the final data structures. You should show proper sharing of data structures, except that you may repeat the symbol `nil` multiple times in your diagram.

```
val t = ("hi", 4.5, "there")
val (x,_,z) = t
val (a::b::cs) = x::[z,z]
val m = [x::a::cs, [], b::"bob"::"sue"::cs, nil]
```



5) Scheme has dynamic typing, while ML has static typing.

a) [3 pts] What is an advantage of Scheme's dynamic typing?

It allows heterogeneous mixes of different types in a list, and also determining lazily where

b) [3 pts] What is an advantage of ML's static typing?

It checks for type errors early, without requiring any testing to find them.

c) [4 pts] If ML didn't have polymorphic types, then its strong, static typing would be unusably strict. C doesn't have polymorphic types, but some people still find it usable. What does C allow that ML doesn't, which makes up for C's lack of polymorphic types?

Casts and related features, which let the programmer work around the limitations of the type system.

6) [4 pts] Why is automatic garbage collection important to ensure type-safety, i.e., a system where no uncaught type errors can happen? In other words, how could a system with explicit allocation and deallocation (like C++'s `new` and `delete`) break type safety?

If the programmer can free memory explicitly, it can create dangling pointers. If the dangling pointer target is later reallocated to a data structure of a different type, the original freed pointer can be used to access data of one type as if it were of a different type.

7) [10 pts] In the MiniML interpreter project, evaluating a tuple expression required recursively evaluating the list of element expressions. Show how to use `map` to perform this evaluation and then build a `TupleValue` containing the result of the `map` invocation, without any helper functions, by providing the body expression of the following `evalExpr` case:

```
...
| evalExpr(TupleExpr(exprs:Expr list),
           env, global_env):Value =


let val values:Value list =
          map(fn(expr)=>evalExpr(expr, env, global_env),
              exprs)
in TupleValue(values) end
```

8) Lists are a basic data structure used in many high-level languages.

a) [3 pts] Why are lists typically manipulated by recursive functions, while arrays are typically implemented by iterative loops?

Lists are a recursive datatype; all recursive data types are easily manipulated with recursive functions having the same recursive structure as the datatype.

Arrays have no recursive structure, so loops over their indices are natural.

b) [3 pts] What two basic operations on ML lists are much faster than the analogous operation on C arrays?

`::` and `tl`

c) [2 pts] What basic operation is much faster on a C array than on an ML list?

random access indexing

9) [10 pts] For the following ML function, illustrate the process of type inference systematically. Show the constraints introduced for each subexpression of the program, and show the final inferred type.

```
fun f(a, b) =
  if null(tl(b)) then a(hd(b)) + 1
  else f(a, tl(b))
```

Types of arguments and result:

```
a:   'a
b:   'b
res: 'res
```

Considering each expression in turn:

```
tl(b):           'b = 'd list
null(tl(b)):     no change
hd(b):           no change
a(hd(b)):        'a = 'd -> 'e
a(...) + 1:      'e = int, 'res = int
tl(b):           no change
f(...):          'a = 'a, 'd list = 'b, 'res = 'res
```

Resulting values for type variables:

```
'a   = 'd -> int
'b   = 'd list
'res = int
```

Resulting type for “f”:

```
f: ('d -> int) -> 'd list -> int
```

Or in pretty form:

```
f: ('a -> int) -> 'a list -> int
```