Name: ***Solutions***

Due Monday, November 2, at the start of class.

Take in a single 4-hour period, self-timed. You may refer to any of your notes, lecture slides, or course readings, but you may not use the computer to experiment or verify your answers.

Your solutions should be written electronically in plain ASCII text in one file and turned in via email to `vass@cs.washington.edu`.

100 points total.

1) Consider the function `remove_prefix` that takes two lists, where the first list is expected to be a prefix of the second list, and returns the remainder of the second list after the first list has been removed. If the first list isn't a prefix of the second list, the function fails. No side-effects should be done. For example:

```
(remove_prefix '(3 (4 5) 6) '(3 (4 5) 6 7 8))  →  (7 8)
(remove_prefix '(3 4 5) '(3 4 5))  →  ()
(remove_prefix '(3 4 5) '(3 4)) €→failure
(remove_prefix '(3 4 5) '(3 4 6 7 8)) €→failure
```

a) [3 pts] Implement the `remove_prefix` function in Scheme, using good Scheme style. Implement failure by returning false.

```
(define (remove_prefix l1 l2)
   (cond ((null? l1) l2)
         ((or (null? l2)
              (not (equal? (car l1) (car l2)))) #f)
         (else (remove_prefix (cdr l1) (cdr l2)))))
```

b) [3 pts] Implement the `remove_prefix` function in ML, using good ML style. Implement failure by raising an exception. What is the type of `remove_prefix`?

```
exception Not_A_Prefix;

fun remove_prefix([], ys) = ys
|   remove_prefix(_ , []) = raise Not_A_Prefix
|   remove_prefix(x::xs, y::ys) =
     if x = y then remove_prefix(xs, ys)
     else raise Not_A_Prefix;

remove_prefix: ''a list * ''a list -> ''a list
```

2) [4 pts] The `reduce` function we've been using implements **right-to-left** reduction, e.g.

```
reduce f base [x1,x2,x3] ⇔ f(x1, f(x2, f(x3,base)))
```

Implement (in either Scheme or ML) a function `reduce_lr` that does **left-to-right** reduction, e.g.

```
reduce_lr f base [x1,x2,x3] ⇔ f(f(f(x1, x2), x3), base)
```

Give an example expression whose behavior when using `reduce_lr` differs from that of `reduce`.

```
fun reduce_lr f base nil = base
|   reduce_lr f base [x] = f(x,base)
|   reduce_lr f base (x1::x2::xs) =
     reduce_lr f base (f(x1,x2)::xs)
```

```
reduce_lr (op /) 1.0 [10.0,2.0,3.0]
```

3) [4 pts] Implement in Scheme a function `abortable-map` that takes a 2-argument function `f` and a list `l` and maps `f` over the elements of `l`. Inside `abortable-map`, `f` is passed each element of the list `l` and an extra continuation argument that, if invoked on a value `v`, immediately stops execution of `abortable-map` and returns `v` from `abortable-map`. You should use `map` in your implementation of `abortable-map`. Here are some sample uses (note my cool `abortable-sqrt` curried helper function!):

```
(define (abortable-sqrt error-val)
  (lambda (num if-error)
    (if (< num 0) (if-error error-val) (sqrt num))))
```

```
(abortable-map (abortable-sqrt #f) '(9 16 25))   → '(3 4 5)
```

```
(abortable-map (abortable-sqrt #t) '(9 -16 25))  → #t
```

```
(define (abortable-map f l)
  (call-with-current-continuation (lambda (exit)
    (map (lambda (num) (f num exit)) l))))
```

4) [4 pts] Implement the ML function `rms` that takes two lists a and b of n reals (in curried fashion) and computes

$$\sqrt{\frac{\sum_{i=1}^{n}((a_i - b_i)^2)}{n}}$$

(The root-mean-square function is used in statistics as one metric of how different two sequences of numbers are.)

You should exploit the `map2` and `reduce` functionals (both of which take 3 arguments in curried fashion; `map2` takes a binary operator and two lists) in your solution. You may assume that the functions `square` and `sqrt` of type `real->real` and the function `length` of type `'a list->int` are available. Also, the expression `real(n)` coerces an integer `n` into a real number.

```
fun rms a b =
   let
      val squares_of_diffs =
         map2 (fn(ai,bi) => square (ai - bi)) a b;
      val sum = reduce (op+) 0.0 squares_of_diffs
   in
      sqrt (sum / real (length a))
   end;
```

5) a) [3 pts] Declare an ML datatype `NTree` for constructing n-ary trees, where each interior node in the tree has a data value (of some unknown type by which the `NTree` is parameterized) and a list of subtrees. (Each node may have 0 or more subtrees, and different nodes may have different numbers of subtrees.) An empty tree is possible, too.

```
datatype 'a NTree =
   Empty |
   Node of 'a * 'a NTree list;
```

b) [4 pts] Implement a functional `map_preorder` that takes a unary function and an `NTree` and returns a flattened list of the results of applying the unary argument function to each of the data values in the tree, in preorder, i.e. the result of the function on the data value for a node appears before the function results for its subtrees, and the subtrees' data values are included in left-to-right sequence. E.g.,

```
map_preorder square (Node (3, [Node(4, []), Node(5, [])]))
   → [9, 16, 25]
```

(FYI, `@` is the infix append operator in ML.)

```
fun map_preorder _    Empty = []
  | map_preorder func (Node(x, trees)) =
     func x ::
       reduce (op@) []
          (map (map_preorder func) trees);
```

c) [5 pts] map_preorder is not a particularly general operation on NTrees: we might wish to program other orders of traversal (e.g. postorder, reverse postorder), and we might wish to perform some more general operation than just accumulating all the data values into a list. Define a more general functional, map_ntree, that is parameterized by the unary element mapping function as in map_preorder, as well as a binary combining function that enables clients to implement their own traversal orders and accumulating methods. Show how your more flexible functional can be used to define map_preorder, and show how it can be used to get other effects.

Here's one possibility which keeps the basic list structure but abstracts the decision of how to order the list:

```
fun map_ntree _        _            Empty = []
  | map_ntree map_func combine_func (Node(x, trees)) =
     let val rest =
          reduce (op@) []
             (map (map_ntree map_func combine_func) trees)
     in
        combine_func(map_func x, rest)
     end;
```

Here's the map_preorder function:

```
fun map_preorder func =
  map_ntree func (fn (elem,rest) => elem::rest);
```

Here's another possibility:

```
fun map_postorder func =
  map_ntree func (fn (elem,rest) => rest@[elem]);
```

d) [4 pts] Define a tree_filter functional that takes a predicate function and an NTree, and returns a list of all those subtrees (including the argument tree itself) that satisfy the predicate. The predicate takes an NTree as input. All subtrees of an NTree are considered; for the always-true predicate, for example, the tree_filter functional will return a list of all *N* subtrees for an input tree with *N* nodes.

```
fun tree_filter _    Empty = []
  | tree_filter pred (n as Node(x, trees)) =
     let val rest =
          reduce (op@) []
             (map (tree_filter pred) trees)
     in
        if pred n then n::rest else rest
     end;
```

e) [3 pts] Explain how lazy evaluation would be useful for clients of `tree_filter` that examined the possible subtrees matching the predicate and selecting the first that satisfied the predicate.

Rather than building a list of all possible matching subtrees and then selecting the first one, a lazy language will only need to execute `tree_filter` until the first match is found, returning that subtree to clients. This has the potential to greatly increase the performance of the client.

6) For the following ML examples, illustrate the process of type inference systematically. Show the constraints introduced for each subexpression of the program, and show the final inferred type of the function.

a) [4 pts]

```
fun frob a b c =
  if null c then a + b
  else frob a 6.7 c


Types of arguments and result:
a:   'a
b:   'b
c:   'c
res: 'res


Considering each expression in turn:
(null c):          'c = 'e list
(a + b):           'a = 'b = <some numeric type> = 'res
(recursive call): 'a = 'a, 'b = real, 'c = 'c, 'res = 'res


Resulting values for type variables:
'a   = real
'b   = 'e list
'c   = real
'res = real


Resulting type for "frob":
frob : real -> real -> 'e list -> real
```

b) [4 pts]

```
fun swizzle a b =
  if null (tl b) then (a (hd b)) + 1
  else swizzle 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
(recursive call): 'a = 'a, 'd list = 'b, 'res = 'res
```

```
Resulting values for type variables:
'a   = 'd -> int
'b   = 'd list
'res = int
```

```
Resulting type for “swizzle”:
swizzle: ('d -> int) -> 'd list -> int
```

c) [4 pts]

```
fun flubber (a, b) = fn c => a (b c)
```

```
Types of arguments and result:
a:   'a
b:   'b
c:   'c
res(flubber):      'res
res(fn c => ...) : 'fres
```

```
Considering each expression in turn:
(b c):          'b = 'c -> 'd
(a (...)):      'a = 'd -> 'e, 'fres = 'e
(fn c => ...): 'res = 'c -> 'fres


Resulting values for type variables:
'a    = 'd -> 'e
'b    = 'c -> 'd
'c    = 'c
'fres = 'e
'res  = 'c -> 'e


Resulting type for "flubber":
flubber : ('d -> 'e) * ('c -> 'd) -> 'c -> 'e

[flubber is the compose function!]
```

7) In Scheme, variables declared using `define` are implicitly mutable; `set!` can change their bindings. Similarly, `car` and `cdr` fields of cons cells are implicitly mutable. In ML, variables declared using `val` or `fun` and the `hd` and `tl` components of cons cells are immutable; explicit references must be used for things that are mutable.

a) [2 pts] What is an advantage of the ML approach?

Code becomes clearer to programmers about where side-effects may happen. Plus the compiler can optimize the code more easily.

b) [4 pts] Translate the following Scheme expressions into ML, using explicit references:

```
(define lst '(3 4))
(set! lst '(5 6))
(set-car (cdr lst) 7)
(define id (lambda (x) x))
```

```
val lst = ref [ref 3, ref 4];
lst := [ref 5, ref 6];
hd (tl (!lst)) := 7;
val id = ref (fn x => x); (* which doesn't typecheck! *)
```

c) [4 pts] Why would it be hard, under ML's typing rules with the restrictions on polymorphic expressions, to make all `val` and `fun` binding implicitly mutable?

Because if all top-level `val` bindings are implicitly to references, which cannot be polymorphic, no polymorphic top-level variables (including function bindings!) can exist.

8) a) [4 pts] Define an ML signature `GROUP` for a mathematical group, i.e. a type `T` with an associative binary operator `Op` and an identity value `Id`.

```
signature GROUP = sig
   type T;
   val Op:T*T->T;
   val Id:T;
end;
```

b) [4 pts] Define ML structures `IntAdd` and `RealMul` that each implement the `GROUP` signature, based on integer addition and real multiplication, respectively.

```
structure IntAdd:GROUP = struct
   type T = int;
   val Op = (op +);
   val Id = 0;
end;

structure RealMul:GROUP = struct
   type T = real;
   val Op = (op * ):real*real->real;
   val Id = 1.0;
end;
```

c) [3 pts] Should the the `IntAdd` and `RealMul` structures be declared to be in the `GROUP` signature using `:` or `:>`? Why?

The `:` symbol should be used, because `GROUP` is defining a view onto an underlying type, not a full-fledged ADT. In particular, there are no constructors of the type `GROUP.T` so encapsulating the value of `T` would make it impossible to use the `IntAdd` and `RealMul` structures.

d) [4 pts] Write an ML functor `Repeatedly` parameterized by a `GROUP` that exports a single function, `repeatedly`, that takes a value of the argument group and a nonnegative integer repeat count `n` and applies the group's operator to the value `n` times (i.e., reducing the operator over `n` copies of the argument value, using the group's identity value if `n` is zero) and returns the result.

```
functor Repeatedly(G:GROUP) = struct
   fun repeatedly v 0 = G.Id
   |   repeatedly v n = G.Op(v, repeatedly v (n-1));
end;
```

e) [3 pts] Instantiate the `Repeatedly` functor for the `IntAdd` and `RealMul` structures. What are more natural names for the `repeatedly` function in each of these instantiations?

```
structure RepeatedlyIntAdd = Repeatedly(IntAdd);
val multiply = RepeatedlyIntAdd.repeatedly;

structure RepeatedlyRealMul = Repeatedly(RealMul);
val exponentiate = RepeatedlyRealMul.repeatedly;
```

9) [3 pts] Why would it be harder to include dynamic scoping in a language like ML than in a language like Scheme?

Static typechecking would be much more difficult under dynamic scoping.

10) a) [4 pts] Suppose ML didn't have lexically-nested first-class functions, but only top-level first-class functions (like C and C++ function pointers). What programming idioms (i.e., typical programming patterns) could still be used, and what idioms would be lost? How important is this language feature, in your opinion?

You could still do all of the standard higher-order list iteration functions like map and filter, but the functions sent as parameters to these functions must have no free variables, thereby greatly reducing the flexibility of this idiom. Currying would become impossible, as would many other applications of returning functions from functions. Idioms like the menu data structure of functions and infinite streams via functions also become hard(er).

b) [4 pts] Suppose ML didn't have first-class lexically-nested functions, but only lexically-nested functions that could be passed as arguments and invoked but not returned as results or stored in data structures (like functions in Pascal and Modula-3). What programming idioms could still be used, and what idioms would be lost? How important is this language feature, in your opinion?

You could still do all of the standard higher-order list iteration functions like map and filter (including the ability to have free variables from the lexical scope in the function parameters) but you lose the ability to make use of currying to partially evaluate a function. Idioms like the menu data structure of functions and infinite streams via functions also become impossible.

11) a) [3 pts] What kinds of Haskell programs are harder or much less efficient to write in ML, due to ML's lack of lazy evaluation?

programs that manipulate streams or other potentially infinite data structures

control-structure-like functions that avoid evaluating some arguments, which would have to be rewritten in ML to use explicit function arguments

b) [3 pts] What kinds of Haskell programs can't be written in ML without substantial code duplication, due to ML's lack of type classes?

programs that use some form of bounded polymorphism, applying to a collection of types, but not all types.

For example, if a Haskell program defines function `length` on the `collection` type class and then uses the same implementation in many subclasses of `collection`, an equivalent ML program would have a copy of function `length` (with a different name) for each data type where it's used.

12) [2 pts] Why do all recursive ML types (lists, trees, etc.) involve a `datatype` declaration (perhaps a built-in one for built-in types like lists)?

A recursive declaration needs at least one recursive case and at least one base case, and a datatype declaration provides the only way to define a type with multiple cases.

13) [4 pts] Why is automatic garbage collection required to have a type-safe system? In other words, how could a system with explicit allocation and deallocation 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, violating type safety.