Name: ____

CSE 341 : Programming Languages
Midterm, Spring 2015

Please do not turn the page until 12:30.

Rules:

- Closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- Please stop promptly at 1:20.
- There are 100 points total, distributed unevenly among 5 questions.
- When writing code, style matters, but don't worry too much about indentation.

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not in order of difficulty. Skip around. Get to all the problems.
- The problems are not all worth the same number of points. Get easier points first.
- If you have questions, ask.
- Don’t worry too much; you’re here to learn. Good luck!

PROBLEM 1. (25 points total) To start off, we'll implement `flatMap` which takes a function of type `'a -> 'b list` and an `'a` list and returns a **single** `'b list`. For example,

```
flatMap (fn x => [x, x]) [1, 2, 3]
```

evaluates to `[1, 1, 2, 2, 3, 3]` and

```
flatMap (fn y => [y, y + 1]) [1, 2, 3]
```

evaluates to `[1, 2, 2, 3, 3, 4]`

(8 points) Please implement `flatMap` below. The implementation does **not** need to be tail recursive, and you may use the list append operator `@`.

```
(* flatMap : ('a -> 'b list) -> 'a list -> 'b list *)


fun flatMap f [] = []
  | flatMap f (x::xs) = f x @ flatMap f xs
```

Next, we'll implement `fold2` which takes a function from type `‘a -> ‘b -> ‘c -> ‘c`, an `‘a list`, and a `‘b list`, and returns a value of type `‘c`. For example,

```
let
     fun helper a b acc =
          if a < b
          then a + acc
          else b + acc
in
     fold2 helper [1, 2, 3] [3, 2, 1] 0
end
```

evaluates to `4` and

```
let
     fun helper a b acc =
          a ^ b ^ acc
in
     fold2 helper [“a”, “b”] [“x”, “y”] “”
end
```

evaluates to `“axby”.` Notice the order in which fold2 processes the lists!

(8 points) Please implement `fold2` below. For full points, please make your implementation tail recursive. (You will still get most of the points for non-tail-recursive implementations.)

```
(* fold2 : (‘a -> ‘b -> ‘c -> ‘c) ->
                ‘a list -> ‘b list -> ‘c -> ‘c *)
```

```
(* tail recursive, wrong order (7 points) *)
fun fold2 f [] [] acc = acc
  | fold2 f (x::xs) (y::ys) acc =
      fold2 f xs ys (f x y acc)

(* not tail recursive, right order (7 points) *)
fun fold2 f [] [] acc = acc
  | fold2 f (x::xs) (y::ys) acc =
      f x y (fold2 f xs ys acc)
```

```
(* tail recursive, right order (8 points) *)
fun fold2 f xs ys acc = let
  fun loop [] [] acc = acc
    | loop (x::xs) (y::ys) acc =
        loop f xs ys (f x y acc)
in
  loop (rev xs) (rev ys) acc
end
```

Finally, we'll implement `inBoth` which takes two `'a lists` and returns a list containing all the elements that appear in both argument lists. For example,

```
inBoth [1, 2, 3] [2, 4, 8]
```

evaluates to `[2]` and

```
inBoth ["a", "b", "c"] ["x", "y", "z"]
```

evaluates to `[]`

To implement `inBoth`, we'll use three helper functions: `filter`, `mem`, and `flip`:

```
fun filter f [] = []
  | filter f (x::xs) =
           if f x
           then x :: filter f xs
           else filter f xs

fun mem x [] = false
  | mem x (y::ys) = (x = y) orelse mem x ys

fun flip f x y = f y x
```

(9 points) Please implement `inBoth` using `filter`, `mem`, and `flip`. The solution should only be one line of code. If you cannot determine how to implement `inBoth` in our limited time using `filter`, `mem`, and `flip`, you can implement it directly for partial credit.

```
fun inBoth xs ys = filter (flip mem ys) xs
```

PROBLEM 2. (15 points total) What are the types of these functions?

```
fun foo f [] e1 e2 = e1
  | foo f [x] e1 e2 = e2
  | foo f (x::xs) e1 e2 = f (x, foo f xs e1 e2)
```

(3 points) foo : `('a * 'b -> 'b) -> 'a list -> 'b -> 'b -> 'b`

```
fun repeat f c x =
     if c x
     then x
     else repeat f c (f x)
```

(5 points) repeat : `('a -> 'a) -> ('a -> bool) -> 'a -> 'a`

```
fun bar (x::xs) (y::ys) (z::zs) = (x, y, z) :: bar xs ys zs
  | bar _ _ _ = []
```

(5 points) bar : `'a list -> 'b list -> 'c list -> ('a * 'b * 'c) list`

```
fun baz a [] = [[]]
  | baz a (x::xs) = map (fn r => a :: x :: r) (baz a xs)
```

(2 points) baz : `'a -> 'a list -> 'a list list`

PROBLEM 3. (25 points total) This question has **five** parts. We treat each part independently, as though it were in its own separate namespace: bindings defined in previous parts are not valid in subsequent parts. For each part, write what `ans` is bound to.

```
val x = 3
fun f y = x
val x = 4
val ans = f 5
```

(6 points) ans is `3`

```
val x = 3
fun f x = x
val x = 4
val ans = f 5
```

(6 points) ans is `5`

```
val add1 = fn (a, b) => (a + 1, b + 1)
val pairify = map (fn x => (x, ~x))
val ans = map add1 (pairify [1,2,3,4])
```

(6 points) ans is `[(2,0),(3,~1),(4,~2),(5,~3)]`

```
val sub = fn x => x - 2
fun sub x = 3 - sub x
val sub = fn x => sub x - 4
val ans = sub 5
```

(7 points) ans is **`~ 4`**

BONUS! (not required)

```
fun f [] = []
  | f (x::xs) = let
      val f = fn x => x :: f xs
    in f xs end
val ans = f [1,2,3,4]
```

(5 points) ans is **`[[2,3,4],[3,4],[4],[]]`**

PROBLEM 4. (15 points total) Tracking references can be subtle. Consider this toy program which outputs 8 integers:

```
val a = ref 0

fun bump x = x + 1

val y = ref bump

fun foo x y z = x (!y + z)

fun printInt i = print (Int.toString i ^ " ")

fun bar x =
    let
      val _ = a := (!y (!a))
      val _ = printInt (!a)
    in
      foo (!y) a x
    end

val _ = printInt (bar 1)
val _ = printInt (bar 1)
val _ = y := foo (!y) a
val _ = printInt (bar 1)
val _ = printInt (bar 1)
```

Hint: It may help to draw pictures of what references point to at different times.

(7 points) What are the first four outputs of this program? Circle the correct option. Some answers are more correct than others and will receive more partial credit.

```
(a)   1    3    4    6

(b)   0    2    3    5

(c)   1    2    3    4

(d)   1    2    2    2

(e)   0    2    0    2
```

`(f)   1    3    2    4`

(8 points) What are the next four outputs of this program? Circle the correct option. Some answers are more correct than others and will receive more partial credit.

(a) 0 5 0 5

(b) 4 6 8 10

(c) 4 11 10 23

(d) 5 12 11 24

(e) 5 11 8 17

(f) 5 7 11 13

PROBLEM 5. (20 points total) Here is an example of an ML module signature:

```
signature STACK = sig
     type ‘a t
     exception Empty
     val empty : ‘a t
     val push  : ‘a -> ‘a t -> ‘a t
     val pop   : ‘a t -> ‘a * ‘a t
end
```

The type `‘a STACK.t` represents a stack of ‘a values.

And here is an example of a module implementing the STACK signature:

```
structure ListStack <: STACK = struct
     type ‘a t = ‘a list
     val empty = []
     fun push x xs = x :: xs
     fun pop [] = raise Empty
         | pop (x :: xs) = (x, xs)
end
```

Note that this implementation satisfies the following two properties:

(A) `pop empty` raises the `Empty` exception

(B) `pop (push x stack)` returns `(x, stack)`

Now, consider this signature:

```
signature NONEMPTY = sig
     type ‘a t
     exception Single
     val single : ‘a -> ‘a t
     val cons : ‘a -> ‘a t -> ‘a t
     val head : ‘a t -> ‘a
     val tail : ‘a t -> ‘a t
end
```

The type `‘a NONEMPTY.t` represents a nonempty sequence of ‘a values.

(17 points) Please provide a module implementing the NONEMPTY signature. Please implement your module directly (i.e. do not call into the Stack module from the example). Your module implementation should satisfy the following properties:

A. `head (single x)` returns `x`
B. `head (cons x xs)` returns `x`
C. `tail (single x)` raises the `Single` exception
D. `tail (cons x xs)` returns `xs`

```
structure NonEmptyList <: NONEMPTY = struct
     type ‘a t = ‘a list
     exception Single
     fun single x = [x]
     fun cons x xs = x :: xs
     fun head [] = raise Single (* unreachable case *)
       | head (x :: xs) = x
     fun tail [] = raise Single (* unreachable case *)
       | tail [x] = raise Single
       | tail (x :: xs) = xs
end
```

(3 points) Can code using your module ever call `head` in a way that triggers an exception? Why or why not?

`head` never raises an exception because no function in the implementation of `NonEmptyList` ever produces an empty list, and client code cannot generate a `NonEmptyList.t` value except using the functions provided by the module.