CSE 341 : Programming Languages
Midterm, Spring 2014

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.
- You can separate pages, but please staple them back together before you leave.
- There are 100 points total, distributed unevenly among 6 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.
- If you have questions, ask.
- Don’t worry too much; you’re here to learn. You are smart and can totally do this!!!

1. **(25 points)** In this question we will use `map` and `fold` over lists to implement `tmap` and `tfold` over variable arity trees. Assume these implementations of `map` and `fold` for lists:

```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
fun map f [] = []
  | map f (h::t) = f h :: map f t

(* fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun fold f base [] = base
  | fold f base (h::t) = f h (fold f base t)
```

Consider this implementation of variable arity trees:

```
datatype 'a tree = Node of 'a * ('a tree list)
```

How should we fill in the blank to map function `f` over an entire tree?

```
(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f (Node (x, ts)) = ______________________________
```

(9 points) Circle the correct way to fill in the blank (only **one** of the options is correct):

```
(e) Node (f x, map (tmap f) ts)
```

How should we fill in the blank to fold function `f` with `base` over an entire tree?

```
(* tfold : ('a -> 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f base (Node (x, ts)) = ______________________________
```

(9 points) Circle the correct way to fill in the blank (only one of the options is correct):

```
(d) f x (fold (fn t => fn acc => tfold f acc t) base ts)
```

(4 points) Use `foldt` and the `add` function below to fill in the blank for `sumt`, a function which adds up all the ints in an int tree. Note that `sumt` uses a *val* binding!

```
fun add a b = a + b
```

```
val sumt = foldt add 0
```

(3 points) Fill in the blank to show the type of `sumt`:

```
sumt :   int tree -> int
```

2. **(15 points)** Rewrite this function to be tail recursive (keep the same order!):

```
fun pairUp x [] = []
  | pairUp x (h::t) = (x, h) :: pairUp x t
```

```
fun pairUp x ys =
   let
     fun loop [] acc = acc
       | loop (h::t) acc = loop t ((x,h)::acc)
   in
     List.rev (loop ys [])
   end
```

Rewrite this function to be tail recursive (keep the same order!):

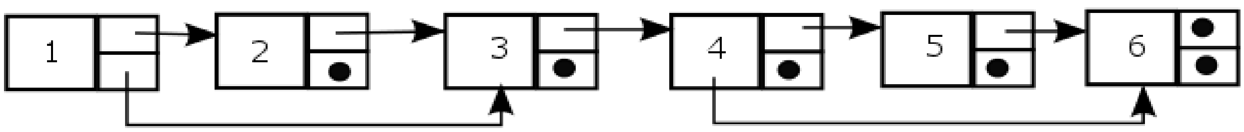
```
fun xprod [] ys = []
  | xprod (x::xs) ys = pairUp x ys @ xprod xs ys
```

```
fun xprod xs ys =
   let
     fun loop [] acc = acc
       | loop (x::xs) acc = loop xs (acc @ (pairUp x ys))
   in
     loop xs []
   end
```

3. **(15 points)** Consider the following datatype:

```
datatype SkipList = Null | Node of int * SkipList * SkipList
```

We can use SkipList to represent lists where we can “skip ahead” to later parts of a list. For example the bindings below represent the following list:




```
val n6 = Node(6, Null, Null)
val n5 = Node(5, n6, Null)
val n4 = Node(4, n5, n6)
val n3 = Node(3, n4, Null)
val n2 = Node(2, n3, Null)
val n1 = Node(1, n2, n3)
```

Consider these two functions which attempt to flatten a SkipList into an ordinary list:

```
fun flatten slist  =
  case slist of
       Node (x, sl1, sl2)  => x :: (flatten sl2)
     | Node (x, sl1, Null) => x :: (flatten sl1)
     | Null => []

exception NullListError

fun flatten_again slist =
  case slist of
       Node(x, sl1, Node(a, b, c)) => x :: (flatten_again sl1)
     | Node(x, sl1, Null)  => x :: (flatten_again sl1)
     | Node(x, Null, Null) => [0]
     | Null => raise NullListError
```

(6 points) Assuming the implementation of `fold` and `add` from Problem #1 and the definitions above, what value will `sum1` be bound to below? If sum1 will fail to evaluate due to an uncaught exception, write the name of the thrown exception in the blank.

```
val sum1 = fold add 0 (flatten n1)
```

sum1 = **4 // (flatten n1) yields the list [1, 3]**

What value will `sum2` be bound to below? If sum1 will fail to evaluate due to an uncaught exception, write the name of the thrown exception in the blank.

```
val sum2 = fold add 0 (flatten_again n1)
```

sum2 = **throws NullListError**

// (flatten_again n6) matches the second pattern
// and makes the recursive call (flatten_again Null)

(5 points) Provide a SkipList built with the `Node` constructor that will cause `flatten_again` to throw an exception, or if that is not possible explain why.

There are many possibilities, here's one:

```
Node(1, Null, Node(2, Null, Null))
```

(4 points) Using all the same lines in `flatten`, but in a different order, write a function `flatten_yet_again` such that `flatten_yet_again n1` evaluates to `[1, 3, 4, 6]`:

```
fun flatten_yet_again slist  =

case slist of
        Node (x, sl1, Null) => x :: (flatten sl1)
      | Node (x, sl1, sl2)  => x :: (flatten sl2)
      | Null => []
```

4. **(15 points)** This question has **three** parts. We treat each part as though it were in its own separate namespace: bindings defined in previous parts are not valid in subsequent parts.

(5 points) Assuming the implementation of `map` from Problem #1, what is ans bound to after this code executes?

```
val (a, b) = (2, 4)
val add1   = (fn x => x + 1)
val times5 = (fn x => x * 5)
val square = (fn x => x * x)

fun f x y z =
  let
    val g = (fn (a, b') => a b)
  in
    y g x
  end

val foo = [(add1, true), (times5, false), (square, true)]
val ans = f foo map (fn x => [x, x])
```

ans = **[5, 20, 16]**

(5 points) Consider the following bindings. What will ans be bound to after this code executes?

```
val (a, b, c, x, y) = (2, 4, 6, 8, 10)
fun f x y =
  (let
    val x = y
    val b = a
    val b = b
  in
    c * b - b
  end) + x + b
val ans = y + f 3 5 - x
```

ans = **19**

(5 points) Consider the following two bindings:

```
fun h f = fn x => f x * f x

val v = h (h (h (fn y => y * y)))
```

Is `v` an int or a function? If it is an int, write its value. If it is a function, write its type and describe what the function computes.

v is a function with type int -> int which raises its argument to the 16th power.

(Optional bonus problem: 3 extra credit points) The bindings below define an int called num, and four functions called f, g, h, and factorial, where factorial is the familiar factorial function. Using each of f, g, h, factorial, and num **exactly once**, write an expression in the blank that will make it so that ans is bound to the factorial of num.

```
val num = 5

fun f a b c d = b a d c
fun g a b c = c a b
fun h a b = b a
fun factorial 0 = 1
  | factorial n = n * factorial (n - 1)
```

ans = **f num g h factorial**

5. **(15 points)** Consider this program:

```
val x = ref 0

fun foo y =
  let
     val _ = x := (!x + 1);
     val _ = print (Int.toString (!x) ^ " ")
  in
     !x + y
  end

val _ = print (Int.toString (foo 1) ^ " ")
val _ = print (Int.toString (foo 1) ^ " ")
val _ = x := 5
val _ = print (Int.toString (foo 1) ^ " ")
val x = ref 10
val _ = print (Int.toString (foo 1) ^ " ")
```

(8 points) What will it print? (Only one option is correct.)

(e) 1 2 2 3 6 7 7 8

Now consider this program:

```
fun bar y =
  let
     val z = ref 0
     val _ = z := !z + 1
     val _ = print (Int.toString (!z) ^ " ")
  in
     !z + y
  end

val _ = print (Int.toString (bar 1) ^ " ")
val _ = print (Int.toString (bar 1) ^ " ")
val z = ref 10
val _ = print (Int.toString (bar 1) ^ " ")
```

(7 points) What will it print? (Only one option is correct.)

(b) 1 2 1 2 1 2

6. **(15 points)** Implement a module satisfying this 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
```

Your implementation should satisfy the following two properties:

(1) `pop empty` should raise the `Empty` exception

(2) `pop (push x stack)` should return `(x, stack)`

(Hint: use lists!)

```
structure stack :> STACK = struct
     type 'a t = 'a list
     exception Empty
     val empty = []
     fun push x xs = x :: xs
     fun pop [] = raise Empty
       | pop (x::xs) =  (x, xs)
end
```