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:

\[
\begin{align*}
(* \text{ map : } (\text{'a} \to \text{'b}) \to \text{'a list} \to \text{'b list} *) \\
\text{fun map } f \ [\] = [] \\
\mid \text{ map } f \ (h::t) = f \ h :: \text{ map } f \ t
\end{align*}
\]

\[
\begin{align*}
(* \text{ fold : } (\text{'a} \to \text{'b} \to \text{'b}) \to \text{'b} \to \text{'a list} \to \text{'b} *) \\
\text{fun fold } f \ \text{base } [\] = \text{base} \\
\mid \text{ fold } f \ \text{base} \ (h::t) = f \ h \ (\text{fold } f \ \text{base} \ t)
\end{align*}
\]

Consider this implementation of variable arity trees:

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

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

\[
(* \text{ tmap : } (\text{'a} \to \text{'b}) \to \text{'a tree} \to \text{'b tree} *) \\
\text{fun tmap } f \ \text{(Node } (x, \ ts)) = \ldots
\]

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

\[
\begin{align*}
(e) \  \text{Node } (f \ x, \ \text{map } \ (\text{tmap } f) \ ts)
\end{align*}
\]
How should we fill in the blank to fold function $f$ with $\text{base}$ over an entire tree?

\[
(* \ \text{tfold} : ('a -> 'b -> 'b) -> 'b -> 'a tree -> 'b *)
\]

fun tfold f base \(\text{Node} \ (x, ts)\) = ________________________________

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

\[
(d) \ f \ x \ \text{(fold} \ \text{(fn} \ t \ => \ \text{fn} \ \text{acc} => \ \text{tfold} \ f \ \text{acc} \ t) \ \text{base} \ ts)\]

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

fun add a b = a + b

val sumt = foldt add 0

(3 points) Fill in the blank to show the type of \text{sumt}:

\[
\text{sumt} : \ \text{int} \ \text{tree} \ \text{->} \ \text{int}
\]
2. (15 points) Rewrite this function to be tail recursive (keep the same order!):

```plaintext
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!):

```plaintext
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:

```plaintext
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:

```plaintext
fun flatten slist =
  case slist of
    Node (x, s11, s12) => x :: (flatten s12)
  | Node (x, s11, Null) => x :: (flatten s11)
  | Null => []

exception NullListError

fun flatten_again slist =
  case slist of
    Node(x, s11, Node(a, b, c)) => x :: (flatten_again s11)
  | Node(x, s11, Null) => x :: (flatten_again s11)
  | 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.

```ocaml
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.

```ocaml
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]`

```ocaml
fun flatten_yet_again slist =
  case slist of
    Node (x, s11, Null) => x :: (flatten s11)
  | Node (x, s11, s12) => x :: (flatten s12)
  | 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 \texttt{map} from Problem #1, what is \texttt{ans} bound to after this code executes?

```plaintext
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 \texttt{ans} be bound to after this code executes?

```plaintext
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:

\[
\text{fun } h \ f = \text{fn } x \Rightarrow f x * f x
\]
\[
\text{val } v = h \ (h \ (\text{fn } y \Rightarrow y * y))
\]

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 \(\text{int} \rightarrow \text{int}\) which raises its argument to the 16th power.

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

\[
\text{val } \text{num} = 5
\]
\[
\text{fun } f \ a \ b \ c \ d = b \ a \ d \ c
\]
\[
\text{fun } g \ a \ b \ c = c \ a \ b
\]
\[
\text{fun } h \ a \ b = b \ a
\]
\[
\text{fun } \text{factorial} \ 0 = 1
\]
\[
\quad | \text{factorial} \ n = n * \text{factorial} \ (n - 1)
\]

\(\text{ans} = f \ \text{num} \ g \ h \ \text{factorial}\)
5. (15 points) Consider this program:

```ocaml
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:

```ml
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:

```ocaml
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!)

```ocaml
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
```