CSE 341 Winter 2019 Midterm

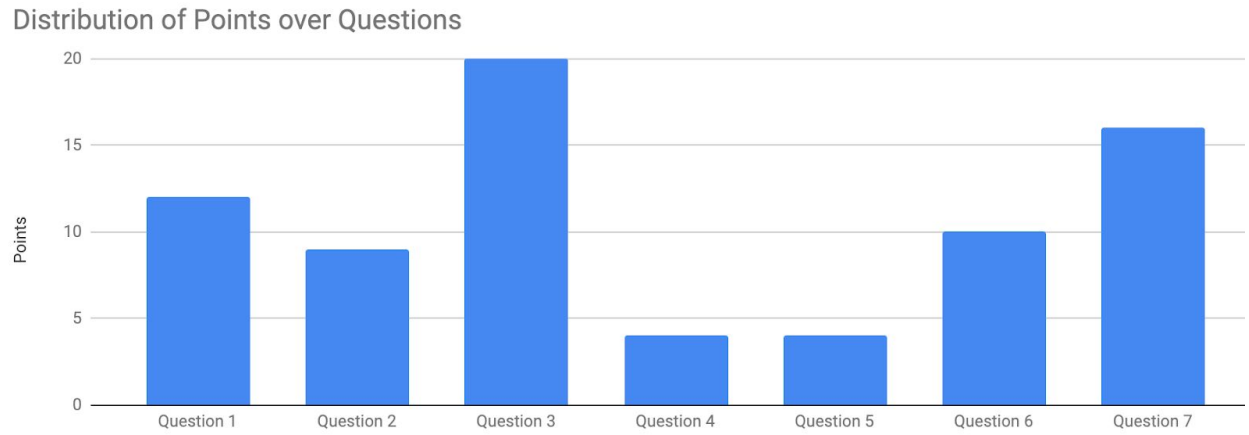
Please do not turn the page until 2:30.

Rules:

- The exam is closed-book, closed-note, etc. except *one side* of a 8.5x11in page.
- **Please stop promptly at 3:20.**
- There are **75 points**, distributed **unevenly** among **7** multi-part questions.
- ***QUESTIONS VARY IN DIFFICULTY. GET EASY POINTS FIRST!!!***
- ***The exam is a bit on long side. Be strategic with your time.***
- The exam is printed double-sided, with pages numbered up to **18**.

Advice:

- Read the questions carefully. Understand before you answer.
- Write down thoughts and intermediate steps so we can give partial credit.
- **Clearly indicate your final answer.**
- Questions are not in order of difficulty. **Always try answering everything.**
- Tear off the Reference Sheet so you can refer to it more easily.
- If you have questions, ask.
- Relax. You are here to learn.



Name : _____ (*please print clearly!*)

QUESTION 1 (12 points). For each of the following SML programs, try to find a way to replace **???** so that `ans` will be bound to **27** after the last line. *If it is impossible to replace ??? so that `ans` is bound to 27, briefly explain why.*

(a, 2 points)

```
val x = ???;
val y = (let x = 27 in x + x end);
val ans = x;
```

Replacement for ??? (or explaination if none possible):

27

(b, 2 points)

```
fun f (x, y) =
    if x < y
    then y - x
    else 2 * y;
val ans = f (???, 27);
```

Replacement for ??? (or explaination if none possible):

0

(c, 2 points)

```
val x = fn x => x * 2;
val ans = x (???);
```

Replacement for ??? (or explaination if none possible):

No, because there 13.5 is not an int.

Name : _____ (*please print clearly!*)

(d, 2 points)

```
val ans = map (fn x => x - 1) ???;
```

Replacement for ??? (or explaination if none possible):

No, because the type of ans will be int list, not int.

(e, 2 points)

```
val ans = foldl (fn (x, y) => x - y) ??? [300, 40, 1];
```

Replacement for ??? (or explaination if none possible):

368

(f, 2 points)

```
val x = 26;
fun foo y z =
    if y <= z
    then x - 1
    else x + 1;
val x = ???;
val ans = foo x x;
```

Replacement for ??? (or explaination if none possible):

No, because the true branch is always taken, giving 27 - 1 = 26.

Name : _____ (*please print clearly!*)

QUESTION 2 (9 points). For each of the following problems assume a fresh set of bindings and:

1. Identify the type of the function `f`
2. Identify the result bound to `ans`
3. Identify whether `f` is tail-recursive (a non-recursive function is trivially not tail recursive). If you think the function is not tail-recursive, explain why (you may mark the code that is specifically violating the tail-recursive property).

Example:

```
(* type of f: int -> int                                          *)
fun f (x) = x + 1;
val ans = f 8; (* ans is bound to: 9                              *)

(* Is f tail-recursive?

No, because f is not recursive.                                   *)
```

(a, 3 points)

```
val z = 8;

(* type of f: _int -> int _____________________________________ *)
fun f x =
  let
    val y = x * 2;
    val x = y - 1;
    val y = x * 2;
  in y + z end;

val z = 10;
val ans = f z; (* ans is bound to: _46_____________________________ *)


(* Is f tail-recursive?


___Not recursive_____________________________________________________ *)
```

Name : _____ (*please print clearly!*)

(b, 3 points)

```
exception E;

(* type of f: int list option -> int                                    *)
fun f zs =
  case zs
    of NONE => raise E
    | SOME [] => raise E
    | SOME (z::[]) => z
    | SOME (z::zs') => z + f (SOME (zs'));

val zs = f (SOME ([3, 4, 1]))
val ans = zs; (* ans is bound to: 8 *)


(* Is f tail-recursive? no - f is not in a tail position in the last case
branch *)
```

(c, 3 points)

```
val a = 42;
fun g (a, b) = b + a;


(* type of f : ___(int * int -> int) -> (int * int)_____________________ *)
fun f y =
    if y (a, a) < 0
    then (y (a, a), ~1)
    else (a, 0);

val ans = f g;   (* ans is bound to: _(42, 0)_____________________________ *)


(* Is f tail-recursive?


_No, it is not recursive_____________________________________________ *)
```

Name : _____ (*please print clearly!*)

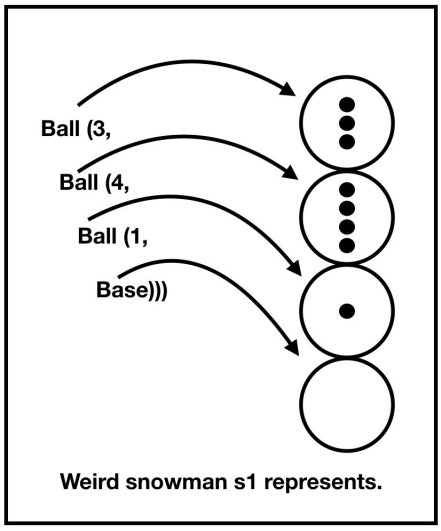
(d, 3 points EXTRA CREDIT -- don't work on this till you're done with everything else!)

```
val g = 12
val x = 5;

(* type of
f : (int * int -> int) list -> (int * int) list -> (int * int) list
*)
fun f gs ys =
  let
    val x = fn y =>
      if y (g, g) < 0 then (y (g, g), ~1) else (g, 0)
  in
    case gs of
       [] => ys
     | g1::[] => [(g, g)]
     | g1::g2::[] => [(x g1)]
     | g1::g2::gs' => (x g2)::(f gs' ys)
  end

val g = fn (x, y) => x - 2;
val h = fn (y, x) => 5 + y;
val ans = f [g, h, h] [(1, 0)];


(* ans is bound to: ___[(12, 0), (12, 12)]_____________________________ *)


(* Is f tail-recursive?


__No, it is not recursive_____________________________________________ *)
```

Name : _____ (*please print clearly!*)

Question 3 (20 points). Consider the following datatype, representing a stack of snowballs (also commonly known as a "snowman").

```
datatype snow = Base | Ball of (int * snow)
```



The `Ball` constructor takes a pair whose first element is an int representing the number of buttons on that snowball and whose second element is another value of type snow "below" that ball in the stack. The bottom of the stack is a Base value (which has no buttons). Here are three examples:

```
val s1 = Ball(3, Ball (4, Ball (1, Base)));
val s2 = Ball(100, Ball (200, Base));
val s3 = Ball(0, Ball (1, Ball (~1, Base)));
```

(a, 5 points) Write a function `remove_buttonless` of type (`snow -> snow`) where the returned `snow` value is similar to the argument but with any `Ball`s having fewer than 1 button removed. If the argument is a `Base`, return `Base`. For example, `remove_buttonless s3` should evaluate to `Ball (1, Base)`.

```
fun remove_buttonless s =
      case s of
            Base => Base
  | Ball (n, b) => if n < 1
                   then remove_buttonless b
                   else Ball (n, remove_buttonless b)
```

Name : _____ (*please print clearly!*)

(b, 5 points) Write a function `build_snowman` of type `((‘a -> int) -> ‘a list -> snow)` such that `(build_snowman f [e1; e2; …; eN])` returns:

```
Ball (f e1, Ball (f e2, ... (Ball (f eN, Base))))
```

For example, `(build_snowman (fn x => x + 1) [3, 4, 1])` should return:

```
Ball (4, Ball (5, Ball(2, Base)))
```

Note that (build_snowman f []) should return Base for any function f. Use recursion directly in your solution. *Do not use any functions from the Reference Sheet in your answer for part (b) here.*

```
fun build_snowman f xs =
    case xs of
        [] => Base
    | x :: xs’ => Ball (f x, build_snowman f xs’)
```

Name : _____ (*please print clearly!*)

(c, 5 points) Now implement `build_snowman` again, but do not use recursion in your function. Instead use functions like rev, append, map, filter, and foldl from the Reference Sheet.

val build_snowman = (* provide your solution below *)

```
fn f => (List.foldl (fn (l, acc) => Ball (f l, acc)) Base) o List.rev
```

Name : _____ (*please print clearly!*)

(d, 5 points) Write a function `interleave` of type (snow -> snow -> snow) which takes as arguments two snow expressions FOO and BAR and evaluates to a snow expression constructed from interleaving each Ball in FOO and BAR, with a Base at the bottom. If the length of s1 and s2 differ, the remaining Ball elements from the longer expression are included as the bottom part of the snowman. The order of elements from both snow expressions should be maintained. For example, given the earlier bindings for s1 and s2 on page 9, `(interleave s1 s2) should evaluate to:`

```
Ball(3, Ball(100, Ball (4, Ball (200, Ball (1, Base)))))
```

```
fun interleave s1 s2 =
  case (s1, s2) of
      (Base, _) => s2
    | (_, Base) => s1
    | (Ball (a1, rest1), Ball (a2, rest2)) =>
       Ball (a1, Ball (a2, interleave rest1 rest2))
```

Name : _____ (*please print clearly!*)

QUESTION 4 (4 points). In this problem, we ask you to give good error messages for why a short ML program does not type-check. A specific phrase or short sentence is plenty. For example, for the program,

```
fun f1 (x,y) = if x then y + 1 else x
```

a fine answer would be, *"the then-branch-expression and the else-branch-expression do not have the same type."* Give good error messages for each of the following:

(a, 2 points)

```
fun g1 x y =
  if x = 0
  then y + 1
  else 2 * g1 (x - 1, y);
```

Your answer:

Calling g1 with a tuple when it expects curried arguments

(b, 2 points)

```
fun g f x =
   case x of
       [] => raise (Fail ":(")
     | [y] => List.hd (f y)
     | x::xs => g f (f x) :: xs
```

Your answer:

For this case statement to be well-typed, we must have `x : A list` for some type `A`. And in branch 2, we apply `f` to an element of `x`, and so `f : A -> B` for some type `B`. But also in branch 2 we apply `List.hd` to the result of `f`, so `B = C list` for some `C`. Thus `g : (A -> C list) -> A list -> C`. But also in branch 3 we cons the result of onto a list of type `A list`, so `A = C list`. Then `g : (C list -> C list) -> (C list) list -> C`. But in branch 3, we pass `(f x)` to `g`, and so we must have `C = (C list) list`, prorsus bananas.

Name : _____ (*please print clearly!*)

QUESTION 5 (4 points). Consider these datatypes:

```
datatype b = PSI | CHI of bool
datatype c = PHI of b | UPSILON of b
datatype d = TAU of d * b | SIGMA of (d -> d) * b
datatype e = RHO of c * b | PI of c list
```

How many distinct *values* are there of each type (e.g., “zero”, “one”, “two”, ..., “infinity”)? Each part is worth 1 point.

b : ____three_____

c : ____six_____

d : _____infinity____

e : _infinity_____

Name : _____ (*please print clearly!*)

QUESTION 6 (10 points). Which of the pairs of expressions are equivalent? Refer to the Reference Sheet for implementations of functions like rev, append, map, filter, and foldl.

In the left column for each row, please write "***Always***" if the expressions are always equivalent, "***Pure***" if the expressions are equivalent when `f` and `g` are pure (always terminate, never throw exceptions, never print, never read or write references, etc.), or "***No***" if the expressions are not equivalent. Remember that `div` is used for integer division in SML. Each part is worth 1 point.

The first three rows are filled out as examples. Please write answers clearly!

Equivalent?	**Expression 1**	**Expression 2**
Always	`x + y`	`y + x`
Pure	`f x + g y`	`g y + f x`
No	`x div y`	`y div x`
Pure	`f(x) + f(x)`	`2 * f(x)`
Always	`f(x + x)`	`f(2 * x)`
Always	`g x orelse f y`	`g x orelse (true andalso f y)`
Always	`(fn (x, y) => f x y) (x, y)`	`f x y`

No	`fun g x =`
	`  let fun f x = x + x`
	`  in f x end`
		`fun g x = fn y => y + y`

Pure	`filter f (append xs ys)`	`let`
		`  val a = filter f ys`
		`  val b = filter f xs`
		`in b @ a end`

No	`rev`	`foldl (fn (acc, x) => acc @ [x]) []`
Always	`rev`	`foldl (fn (acc, x) => x :: acc) []`
No	`filter f (map f l)`	`map f l`
No	`filter f (map f (map f l))`	`map f l`

Name : _____ (*please print clearly!*)

QUESTION 7 (16 points). Consider the `NONEMPTYLIST` signature and its implementation `NonEmptyList`, found on the Reference Sheet. *Each part is worth 4 points.*

The type `non_empty_list` represents a non-empty list of strings. The type `index` represents possible list indices (non-negative integers). The implementation in `NonEmptyList` uses list operations without checking for empty lists.

In any code you write, assume you have access to a variable `x` of type `non_empty_list` bound to `(NonEmptyList.make "foo")`. **WRITE CLEARLY** (if we can't read it, no credit)

(A) Can you cause `NonEmptyList.hd` to throw an Empty list exception? If so, give a concrete program which causes this exception. If not, explain why.

No, because non_empty_list does not give its actual type in the signature, so users must use the make function, which cannot make an empty list. Hd only throws an exception on an empty list.

(B) Can you cause `NonEmptyList.tl` to throw an Empty list exception? If so, give a concrete program which causes this exception. If not, explain why.

No, because non_empty_lists does not give its actual type in the signature, so users must use the make function, which cannot make an empty list. Tl only throws an exception on an empty list.

(C) Can you cause `NonEmptyList.get` to throw an Empty list exception? If so, give a concrete program which causes this exception. If not, explain why.

Yes, because the signature exposes the type of list_index, so users can pass -1 as an index and it will try to take the tl of an empty list.

(D) Explain how to fix the definition of `NONEMPTYLIST` to rule out any errors identified above.

To fix this problem, just hide the actual type of list_index in the signature so that users cannot make a list_index without using to_index (which prevents negatives)

Name : _____ (*please print clearly!*)

EXTRA CREDIT. Consider the following datatype:

```
datatype pr_tree = Leaf
                 | Node of string * (int -> int) * pr_tree * pr_tree
```

This type can be used to build a binary tree of pairs mapping strings to functions. Note that Leafs hold no values. Below are three examples of `pr_trees`:

```
fun f x = x - 1;
fun g x = x + 1;
fun h x = x * 2;
val t1 = Node ("a", g, Node ("b", g, Leaf, Node ("c", g, Leaf, Leaf)), Leaf);
val t2 = Node ("a", f, Node ("b", g, Leaf, Node ("c", h, Leaf, Leaf)), Leaf);
```

(a, 2 points EC) Write a function `pr_fold` of type `(pr_tree * int -> int)` which returns the result of applying the function held by the topmost root of the first argument and its left-most children descendents (all the way to its leftmost Leaf) starting with the second argument. If the first argument is a Leaf, return the value of the second argument. For example:

```
val res1 = pr_fold (t1, 0); (* evaluates to 2 *)
val res2 = pr_fold (t2, 0); (* evaluates to 0 *)
```

```
fun pr_fold (t, acc) =
      Case t of
            Leaf => acc
        |  Node (_, f, t1, _) => pr_fold (t1, f acc)
```

Name : _____ (*please print clearly!*)

(EXTRA CREDIT continued)

(b, 2 points EC) Write a function `get_strings_returning` of type `(pr_tree -> int -> int -> string list)` returning strings from Nodes in the first argument which hold functions that return the third argument when called with the second. The order of the resulting list does not matter. Don't worry about any duplicate strings in pr_trees.

```
fun get_strings_returning t a b =
    case t of
      Leaf => []
     |  Node (s, f, t1, t2) =>
           (if f a = b
           then s :: (get_strings_returning t1 a b)
           else get_strings_returning t1 a b)
                @ (get_strings_returning t2 a b)
```

Reference Sheet

```
fun rev xs =
  let
      fun loop acc l =
        case l of [] => acc
                | h :: t => loop (h :: acc) t
  in
    loop [] xs
  end

fun append xs ys =
  case xs of [] => ys
           | x :: xs' => x :: append xs' ys

fun map f xs =
  case xs of [] => []
           | x :: xs' => f x :: map f xs'

fun filter f xs =
  case xs of [] => []
           | x :: xs' => if f x
                         then x :: filter f xs'
                         else filter f xs'

(* NOTE: this foldl is curried, but the function f it takes is not *)
fun foldl f acc xs =
  case xs of [] => acc
           | x :: xs' => foldl f (f (acc, x)) xs'


(* the "pipeline operator" *)
infix !>
fun x !> f = f x
```

Name : _____ (*please print clearly!*)

```
signature NONEMPTYLIST = sig                        (* For Question 7 *)
    type non_empty_list

    type list_index = int

    val hd: non_empty_list -> string

    val tl: non_empty_list -> string list

    val cons: string * non_empty_list -> non_empty_list

    val make: string -> non_empty_list

    val get: list_index * non_empty_list -> string option

    val to_index: int -> list_index option
end

structure NonEmptyList :> NONEMPTYLIST = struct
    type non_empty_list = string list

    type list_index = int

    fun hd xs = List.hd xs

    fun tl xs = List.tl xs

    fun cons (x, xs) = x::xs

    fun make x = [x]

    fun get (idx, l) =
        if idx >= List.length l then
            NONE
        else let
            fun recurse (i,l) =
                 if i = 0
                 then SOME (hd l)
                 else recurse (i-1, List.tl l)
        in
            recurse (idx, l)
        end

    fun to_index i = if i < 0 then NONE else SOME i

end
```