Name: ____ Student ID #: ________________

CSE 341 Summer 2019 Midterm Exam

July 26, 2019

Do not turn the page until you are instructed to do so.

Rules/Guidelines:

- You must stop working **promptly** when time is called at 1:00pm. Any modifications to your exam (writing *or* erasing) will result in a penalty.
- This exam is closed-book, closed-note, and closed-device with the exception of *one side* of *one* 8.5x 11" piece of paper.
- There are **100** points distributed unevenly among **7** multi-part questions, plus one extra credit question.
- The exam is printed double-sided, with **12** numbed pages. Page 2 is intentionally blank.
- If you abandon one answer and write another, *clearly cross out* the answer(s) you do not want graded. When in doubt, we will grade the answer that appears nearest to the question text.
- If you write an answer on scratch paper, please both *clearly label* which question you are answering on the scratch paper and also *clearly indicate* on the question page that your answer is on scratch paper. Staple all scratch paper to the *end* of the exam before turning in.

Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Feel free to skip around. Be sure you are able to at least attempt every question.
- Work on easier questions first, including the easiest parts of each question. Do not feel the need to answer all parts of one question before moving on to another. You can always circle back.
- Do not attempt the extra credit until you have completed the rest of the exam. It is challenging and will not be worth a lot of points.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- Write down any thoughts or intermediate steps so we can award partial credit, but be sure to clearly indicate your final answer.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions at once.
- Relax. You are here to learn.

Name: ____ Student ID #: ________________

1. Consider the following datatype, where a value of type `race` describes a track and field race:

```
datatype race =
    Sprint of int
    | Hurdles of int * int
    | Relay of race * race
```

Values of this type can be interpreted as follows:

- A value `Sprint i` represents a simple race of `i` meters.
- A value `Hurdles (n,d)` represents a race of `n` hurdles with `d` meters separating each pair of hurdles as well as the first hurdle from the starting line and the last hurdle from the finish line. (You may assume that all hurdles races include at least one hurdle.)
- A value `Relay (r1,r2)` represents a race consisting of the race `r1` followed immediately by the race `r2`.

a) Write a function `total_distance` of type `race -> int` that computes and returns the total distance of the argument race. You should assume that hurdles add no additional distance to a race other than the distance between them.

```
fun total_distance r =
    case r of
     Sprint n => n
     | Hurdles (n, d) => n * d + d
     | Relay (r1, r2) => total_distance r1 + total_distance r2
```

b) Write a function `remove_hurdles` of type `race -> race` that returns a new race of the same total distance and number of segments as the argument race, but with segments of hurdles replaced by a sprint of the same total distance.

```
fun remove_hurdles r =
    case r of
     Hurdles (n, d) => Sprint (n * d + d)
     | Relay (r1, r2) => Relay(remove_hurdles r1, remove_hurdles r2)
     | _ => r
```

c) Consider the following additional datatype, where a value of type `run` describes the actions a runner might take to complete a race:

```
datatype run =
       Run of int
     | Jump
```

Values of this type can be interpreted as follows:

- A value `Run i` means run i meters.
- A value `Jump` means jump over a hurdle.

Write a function `run_race` of type `race -> run list` that returns a list containing a sequence of actions a runner would take to complete the argument race. You may use the `@` function if you wish.

```
fun run_race r =
    case r of
      Hurdles (0, d) => []
     | Hurdles (n, d) => (Run d)::(Jump)::run_race(Hurdles(n - 1, d))
     | Sprint n => [Run n]
     | Relay (r1, r2) => (run_race r1) @ (run_race r2)
```

Name: ____ Student ID #: ________________

2. Consider the following function:

```
fun mystery (xs, ys) =
    case (xs, ys) of
       ([], []) => 0                              (* 1 *)
     | ([], y::ys') => 1 + mystery(xs, ys')       (* 2 *)
     | (x::xs', ys) => 1 + mystery(xs', ys)       (* 3 *)
```

a) What is the type of `mystery`?

`mystery : 'a list * 'b list -> int`

b) What does each of the following calls evaluate to?

i. `mystery ([], [])` **= 0**

ii. `mystery ([], [1, 2, 3])` **= 3**

iii. `mystery ([1, 2, 3, 4, 5], [])` **= 5**

iv. `mystery ([1, 2], [1, 2, 3, 4])` **= 6**

c) Describe in 1-2 English sentences what `mystery` computes.

Computes the sum of the lengths of the two argument lists.

d) Suppose we make each of the following changes to mystery. Which of the following would be true after the change is made? (Circle one option for each change.)

A. `mystery` no longer type-checks or gives a non-exhaustive match warning
B. `mystery` still type-checks without warnings, but now gives a different result in at least one case
C. `mystery` still type-checks without warnings and gives the same result in all cases

Consider each change independently, and ignore the syntactic issue of adding or removing pipe characters (`|`) as necessary.

A	B	**C**	i. Line 2 is moved before line 1
A	B	**C**	ii. Line 3 is moved before line 2
A	B	C	iii. Line 2 is removed
A	B	C	iv. Line 2 is removed and line 3 is replaced with the following: `| (x::xs', y::ys') => 2 + mystery(xs', ys')`

Name: ____ Student ID #: ________________

3. For each of the following programs, if the program has a syntax error or does not type-check, give a *short, specific* sentence indicating what the error is. If the program does type-check, give the type of `f` and the value bound to `ans` after the program runs. Consider each part as a separate program. All programs use the following datatype:

```
datatype greek = Alpha | Beta of string | Gamma of int * greek
```

a)
```
fun f (x, y) =
    if x < y
    then Beta "up"
    else if x > y
    then Beta "down"
    else Alpha
val ans = f (5, 2)
```

f : (int * int) -> greek
ans = Beta "down"

b)
```
fun f2 x =
    case x of
        Alpha => "A"
      | Beta => "B"
      | Gamma => "G"
val ans = f2 Alpha
```

Does not type check: Beta and Gamma are constructors and need arguments

c)
```
fun f g x =
    let
        val (n, s) = g x
    in
        Gamma (n, Beta s)
    end
val ans = f (fn x => (5, "foo")) 9
```

f : ('a -> (int * string)) -> 'a -> greek
ans = Gamma (5, Beta "foo")

d)
```
fun f xs =
case xs of
     [] => Alpha
   | x::[] => Beta  x
   | x::xs' => Gamma (x, f xs')
val ans = f [1, 2, 3]
```

Does not type check: The elements of xs cannot be both strings (to be passed to Beta) and ints (to be passed as the first argument to Gamma).

Name: ____ Student ID #: ________________

4. Consider the following function:

```
fun foldl_if_true f g acc xs =
    case xs of
      [] => acc
     | x::xs' => if f x
                 then foldl_if_true f g (g(x, acc)) xs'
                 else foldl_if_true f g acc xs'
```

a) What is the type of `foldl_if_true`?

`foldl_if_true : ('a -> bool) -> ('a * 'b -> 'a) -> 'b -> 'a list -> 'b`

b) Recall the following functions:

```
List.foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

List.filter : ('a -> bool) -> 'a list -> 'a list
```

Use these functions to write a one-line alternate definition of `foldl_if_true` that is equivalent to the above definition.

`fun foldl_if_true f g acc xs = List.foldl g acc (List.filter f xs)`

c) Use a `val` binding and partial application of `foldl_if_true` to write a one-line definition of `sum_evens`, which has type `int list -> int` and returns the sum of all the even numbers in its argument. (Recall that ML uses the operator `mod` for modulo.)

`val sum_evens = foldl_if_true (fn x => x mod 2 = 0) (fn (x, y) => x + y) 0`

Name: ____ Student ID #: ________________

5. This problems asks you to write three versions of the same function.

a) Write a function `ones_digits` of type `int list -> int list` that returns a list containing the ones digit of each integer from the original list. *Do not* use any helper functions or any library functions besides `::` and `mod`. You may assume all numbers in the argument list are non-negative.

```
fun ones_digits xs =
    case xs of
        [] => []
      | x::xs' => (x mod 10)::(ones_digits xs')
```

b) Write a second version of `ones_digits` that is tail-recursive. *Do not* use any library functions besides `::`, `mod`, and `rev`, though you may use one or more helper functions.

```
fun ones_digits_tail xs =
    let
        fun loop (xs, acc) =
            case xs of
                [] => acc
              | x::xs' => loop(xs', (x mod 10)::acc)
    in
        List.rev (loop(xs, []))
    end
```

c) Recall the following function:

```
List.map : ('a -> 'b) -> 'a list -> 'b list
```

Write a third version of `ones_digits` using a `val` binding and a partial application of `List.map`.

```
val ones_digits_maps = List.map (fn x => x mod 10)
```

Name: ____ Student ID #: ________________

6. For each of the following pairs of expressions, indicate whether the expressions are **ALWAYS** functionally equivalent, **NEVER** functionally equivalent, or functionally equivalent only when `f` and `g` are both **PURE** functions (i.e. they always terminate, never throw exceptions, and have no side-effects). You may assume that `f` and `g` always have the same type but are not the same function, and that all expressions type-check

Expression 1	**Expression 2**	**Equivalent?**
`f x + g x`	`g x + f x`	PURE
`f x orelse g x`	`if f x then g x else false`	NEVER
`fun h x = f (g x)`	`val h = f o g`	ALWAYS

`(f x, g x)`	`let`
	`    val z = g x`
	`    val y = f x`
	`in`
	`    (y, z)`
	`end`
		PURE

`List.filter f (List.map g xs)`	`List.filter (f o g) xs`	NEVER

Name: ____ Student ID #: ________________

7. This problem considers three ML modules `ModTwo1`, `ModTwo2`, and `ModTwo3`, and a signature `MODMATH`, shown on the next page. (You may find it useful to know that, in ML, `val one = ~1 mod 2` binds `one` to the value `1`. This is different from how that expression evaluates in Java.)

   a) Does each of the three modules type-check? (Circle "Y" or "N" for each module.)

      `ModTwo1`: **Y** / N `ModTwo2`: **Y** / N `ModTwo3`: **Y** / N

   b) If possible, fill in each blank so that `res` evaluates to a value other than 0 or 1. If this is not possible, or if the module does not type-check, write "impossible" in the blank instead.

      `val res = ModTwo1.`_**impossible**________________________

      `val res = ModTwo2.`_**impossible**________________________

      `val res = ModTwo3.`_**impossible**________________________

   Now, suppose we replaced the line `type modInt` in the signature `MODMATH` with the line `type modInt = int`.

   c) Does each of the three modules type-check? (Circle "Y" or "N" for each module.)

      `ModTwo1`: **Y** / N `ModTwo2`: **Y** / N `ModTwo3`: Y / **N**

   d) If possible, fill in each blank so that `res` evaluates to a value other than 0 or 1. If this is not possible, or if the module does not type-check, write "impossible" in the blank instead.

      `val res = ModTwo1.`**fromInt 2**___________________________

      `val res = ModTwo2.`**toInt 2**_____________________________

      `val res = ModTwo3.`__**impossible**_______________________

   Now, suppose we replaced the line `val add : modInt * modInt -> modInt` in the signature `MODMATH` with the line `val add : modInt * modInt -> int`.

   e) Does each of the three modules type-check? (Circle "Y" or "N" for each module.)

      `ModTwo1`: **Y** / N `ModTwo2`: **Y** / N `ModTwo3`: Y / **N**

   f) If possible, fill in each blank so that `res` evaluates to a value other than 0 or 1. If this is not possible, or if the module does not type-check, write "impossible" in the blank instead.

      `val res = ModTwo1.`**add(ModTwo1.fromInt 1, ModTwo1.fromInt 1)**__

      `val res = ModTwo2.`__**impossible** ______________________

      `val res = ModTwo3.`__**impossible**_______________________

Name: ____ Student ID #: ________________

```
signature MODMATH =
sig
    type modInt
    val fromInt : int -> modInt
    val toInt : modInt -> int
    val add : modInt * modInt -> modInt
    val subtract : modInt * modInt -> modInt
end
```

```
structure ModTwo1 :> MODMATH =
struct

type modInt = int

fun fromInt n = n
fun toInt n = n mod 2

fun add (x, y) = x + y
fun subtract (x, y) = x - y

end
```

```
structure ModTwo2 :> MODMATH =
struct

type modInt = int

fun fromInt n = n mod 2
fun toInt n = n

fun add (x, y) = (x + y) mod 2
fun subtract (x, y) = (x - y) mod 2

end
```

```
structure ModTwo3 :> MODMATH =
struct

datatype modInt = ZERO | ONE

fun fromInt n = if n mod 2 = 0 then ZERO else ONE
fun toInt n = case n of ZERO => 0 | ONE => 1

fun add (x, y) =
    case (x, y) of
      (ZERO, ZERO) => ZERO
      | (ONE, ONE) => ZERO
      | _ => ONE

val subtract = add

end
```

8. **EXTRA CREDIT – DO NOT ATTEMPT THIS PROBLEM UNTIL YOU HAVE COMPLETED THE REST OF THE EXAM**

Consider the following definitions:

```
datatype Yo = Y of (Yo -> Yo)

fun cool y =
  case y of
    Y f => f y
```

Fill in the blank below so that evaluating the binding of uncool will not terminate. You may *not* use any library functions or create any additional function bindings.

```
val uncool = __cool (Y cool)__________________________
```