Name: _____ NetID: _____

CSE 341 Winter 2020 Midterm

Please do not turn the page until 9:30.

Rules:

- **Please print your name and NetID above extremely clearly.**
- The exam is closed-book, closed-note, etc., except *one side* of a 8.5x11in page.
- **Please stop promptly at 10:20.**
- There are **100 points**, distributed **evenly** among **5** multi-part questions.
- The exam is printed double-sided, with pages numbered up to **12** (then bonus).

Advice:

- Read the questions carefully. Understand before you answer.
- ***Be strategic with your time. Don't get stuck. Get all the points you can.***
- 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.

Question 1 (20 points). For each of part of this question:

1. Identify the type of the function `f`
2. Identify the result bound to `ans`
3. Identify whether f is tail-recursive ("TR" for short). A non-recursive function is trivially not TR. If you think the function is not TR, briefly explain why.

(A)
```
fun f (i, x) =
    if i = 0 then "#" else
        x ^ " " ^ f (i - 1, x ^ x)

val ans = f (3, "ab")
```

f :	int * string -> string
ans =	"ab abab abababab #"
f TR?	No, because f is called within a string concatenation in the else branch

(B)
```
fun f (x, y) = let
    fun g z = let val z = y in z - x end
in
    if x <= 0
    then g (~y * 2)
    else f (y, g x)
end

val ans = f (5, 6)
```

f :	int * int -> int
ans =	f (5, 6) = f (6, 1) = f (1, ~5) = f (~5, ~6) = ~1
f TR?	Yes, because the only recursive call is in an else branch, which is in tail position

(C)
```
fun f g x =
      case x of
        SOME (y :: ys) => g y orelse f (fn z => not (g z)) (SOME ys)
      | _              => false
val ans = f (fn x => x >= 10) (SOME [~5, 24, 9, 10, 7, ~19])
```

f :	('a -> bool) -> 'a list option -> bool
ans =	true
f TR?	Yes, because the only recursive call is in the second argument of an orelse, which is in tail position

(D)
```
fun f g x b = let
     val (h, y) = g x
   in
     case h of
       SOME h' => h' (f g y b)
     | NONE    => b
   end

val ans = f (fn x => case x of
                       y :: ys => (SOME (fn z => y + z), ys)
                     | []      => (NONE, []))
            [1, 2, 3, 4, 5]
            0
```

f :	('a -> ('b -> 'b) option * 'a) -> 'a -> 'b -> 'b
ans =	1 + 2 + 3 + 4 + 5 = 15
f TR?	No, because the recursive call in the first branch of the case expression is inside a function call

Question 2 (20 points).

Consider the following (incomplete) datatype and function:

```
datatype dt = ???

fun foo f t =
  case t of
       A x => if null x then 0 else f (hd x) + foo f (A (tl x))
     | B x => if f x < 0 then f x else f (~x)
     | C (x, y) => if isSome x andalso y x > 0 then valOf x else 0
     | D (x, y) => f x + foo f y
```

(A) Complete the datatype definition for `dt` such that the `foo` function would type-check.

```
datatype dt =

        A of int list

      | B of int

      | C of int option * (int option -> int)

      | D of int * dt
```

(B) What is the type of `foo`?

```
(* your answer here *)
(int -> int) -> dt -> int
```

(C) Mark “T” for each of the bindings below that evaluate to 341 and “F” otherwise.

Binding	**= 341? T/F**
`val a = foo (fn x => x - 1) (A(342))`	**F**
`val b = foo (fn x => x - 1) (A(300, 40, 1))`	**F**
`val c = foo (fn x => x - 1) (B(342))`	**F**
`val d = foo (fn x => ~x) (B(~341))`	**F**
`val e = foo (fn x => ~x) C(SOME 341, SOME 1)`	**F**
`val f = foo (fn x => ~x) (D(~341, A([])))`	**T**

(D) Mark “T” for each of the bindings below that evaluate to 1 and “F” otherwise.

Binding	**= 1? T/F**
`val a2 = foo (fn x => x * x) (A[1])`	**T**
`val b2 = foo (fn x => ~x) (A([A([1])]))`	**F**
`val c2 = foo (fn x => x - 1) (A([342, 1]))`	**F**
`val d2 = foo (fn x => x * x) (A[1, ~1, 0])`	**F**
`val e2 = foo (fn x => x * x) (B(1))`	**T**
`val f2 = foo (fn x => x + 1) (C(NONE, SOME 0))`	**F**

Question 3 (20 points).

(A) For each datatype definition below, write the number of values that have that type. For example,

```
datatype bool = True | False
```

has 2 values (`True` and `False`), while

```
datatype nat = Z | S of nat
```

has infinitely many values (`Z, S Z, S (S Z), S (S (S Z)), ...`). For this question, assume there are infinitely many values of type `int`.

Datatype Definition	**# values**
`datatype foo =` `  A | B | C of foo * foo`	**Infinite**
`datatype bar =` `  A of bar | B of bar | C of bar * bar`	**0**
`datatype baz =` `  A of baz | B of baz | C of baz list`	**Infinite**
`datatype qux =` `  A of bool | B of unit | C of bool * bool`	**7**
`datatype waldo =` `  A of int`	**Infinite**
`datatype weird =` `  A of (int -> int)`	**Infinite**
`datatype huh =` `  A of (huh -> huh)`	**Infinite**

(B) For each pair of bindings `b1, b2` in the rows below, in the rightmost column mark:

- **E** if the bound values are always equivalent
- **P** if the bound values are equivalent for pure arguments (no side effects)
- **N** if the bound values are not equivalent

b1	b2	E / P / N
`fun f x y =` `  x + y`	`val f =` `  fn x => fn y => y + x`	**E**
`fun f (x, y) =` `  x + y`	`val f =` `  fn x => fn y => y + x`	**N**
`fun f x y =` `  if x then y else true`	`fun f x y =` `  x orelse y`	**N**

`fun f g x = let`
`  val a = g x + g x`
`  val b = g 1`
`in`
`  a + b`
`end`
	`fun f g x = let`
	`  val a = g 1`
	`  val b = g x + g x`
	`in`
	`  a + b`
	`end`
		P

`fun f x y z =`
`  if x then y else`
`  if z then x else`
`  true`
	`fun f x y z =`
	`  (x andalso y) orelse`
	`  (z andalso x)`
		N

`fun f x y =` `  f (x + y)`	`fun f x y =` `  f x + f y`	**N**

`fun f x = let`
`  fun id x = x`
 	
`  fun compose f g x = f (g x)`
 	
`  fun flip f x y = f y x`
 	
`  fun curry f x y = f (x, y)`
 	
`  fun uncurry f (x, y) = f x y`
`in`
`  (compose uncurry`
`  (compose flip curry)) id x`
`end`
	`fun f (x, y) =`
	`  (y, x)`
		E

Question 4 (20 points).

(A) For an implementation of `fold` to be correct, it must at least type-check, process every element of its input, and return the accumulator. For each potential variant of `fold` below, put a "T" in the right column if it is correct and "F" otherwise. Assume each variant is independent (i.e., entered into a fresh REPL).

Candidate fold implementation	**Correct? T/F**

`fun fold (xs, acc, f) =`
`  if null xs then acc else`
`  fold (tl xs, f (hd xs, acc), f)`
	T

`val fold =`
`  fn (xs, acc, f) =>`
`  if null xs then acc else`
`  fold (tl xs, f (hd xs, acc), f)`
	F

`fun fold xs f acc =`
`  case xs of`
`  [] => acc`
`  | x :: xs' => fold f (f x acc) xs'`
	F

`fun fold f acc xs =`
`  case xs of`
`  [] => acc`
`  | x :: xs' => fold f (f x acc) xs'`
	T

`fun fold xs f acc =`
`  case xs of`
`  [] => acc`
`  | x :: xs' => fold (xs', f, f x acc)`
	F

`fun fold acc xs f =`
`  case xs of`
`  x :: xs' => f (x, fold acc xs' f)`
`  | [] => acc`
	T

`fun fold (xs, acc, f) =>`
`  if null xs then acc else`
`  fold (xs, f (hd xs, acc), f)`
	F

`fun fold f acc xs =`
`  case xs of`
`  [] => acc`
`  | x :: xs' => fold f (f (hd xs) acc) (tl xs')`
	F

(B) Is it possible to implement `fold` using `map` and `filter`? If so, briefly explain how you would do it. If not, briefly explain why not. (1-2 sentences)

No, because map and filter each take a list and return a list, meaning any combination of the two would have to return a list rather than our accumulator.

(C) Is it possible to implement `map` and `filter` using `fold`? If so, briefly explain how you would do it. If not, briefly explain why not. (1-2 sentences)

Yes. Fold once, accumulating a list of results of the function applied to each list element. Next, fold again to reverse the order of the list so that the mapping of the first element appears first.

Question 5 (20 points).

Consider the SML module `OE` below for working with words in Old English (a language spoken in England from about the 5th century to the 11th, also known as Anglo-Saxon). The Old English alphabet contains three non-ASCII characters: æ (“ash”), ð (“eth”), and þ (“thorn”); it also does not contain the characters j, k, q, v, or z. The module encodes letters in Old English as either strings or numbers. Alliteration was very important in Old English poetry, so `OE` has a function that checks if two words start with the same letter.

```
signature OLDENGLISH = sig
    datatype OELetter = OEStr of string | OENum of int
    type OEWord = OELetter list
    exception NotOLDENGLISH
    val letter_equal : OELetter * OELetter -> bool
    val is_alliterative : OEWord * OEWord -> bool
end

structure OE :> OLDENGLISH = struct
  datatype OELetter = OEStr of string | OENum of int
  type OEWord = OELetter list
  exception NotOLDENGLISH
  val oe_alphabet =
    [ "a","ash","b","c","d","eth","e","f","g","h","i","l"
    , "m","n","o","p","r","s","t","thorn","u","x","w","y" ]

  fun str_of_oenum i =
    (* List.nth (xs, i) returns the ith element of xs
     * or raises an exception if xs does not have an ith element *)
    List.nth (oe_alphabet, i)

  fun letter_equal (l1, l2) =
    case (l1, l2) of
      (OEStr s, OENum n) => s = str_of_oenum n
    | (OENum n, OEStr s) => s = str_of_oenum n
    | _ => l1 = l2

  fun is_alliterative(w1, w2) =
    letter_equal (List.hd w1, List.hd w2)
end
```

(A) Write a value for `word2` of type `OE.OEWord` that will cause `is_alliterative` to raise an exception.

```
val hwaet = [OEStr "h", OEStr "w", OEStr "ash", OEStr "t"]
val word2 =
```

```
(* your code here *)
[] or [OENum 100] (any out of bounds index)
```

```
is_alliterative (hwaet, word2)
```

(B) It is not possible to bind a value to `letter2` of type `OE.OELetter` that will cause the call to `letter_equal` below to raise an exception. Why not?

```
val thorn = OENum 19
val letter2 = ???
val _ = letter_equal (thorn, letter2)
```

```
(* your explanation here *)
Not possible because a number would be equality tested and a
string would tested against the string "thorn"
```

(C) Although it won't cause exceptions in our current set of functions, it's possible to create an `OELetter` that should not be part of the Old English character set, such as `OEStr "k"`. Write a function that will take a string argument and return only a valid `OELetter`.

```
fun make_OELetter_from_string str =
```

```
(* your code here *)
if List.exists (fn x => x = str) oe_alphabet then OEStr str
else raise NotOLDENGLISH
```

(D) Assume now that `OE.OENum` should only be an internal representation and not exposed to the clients. Write a new signature for the module that hides `OE.OENum`.

```
signature OLDENGLISH = sig
```

```
(* your code here *)
type OELetter
val OEStr: string -> OELetter
type OEWord = OELetter list
```

```
  exception NotOLDENGLISH
  val letter_equal : OELetter * OELetter -> bool
  val is_alliterative : OEWord * OEWord -> bool
end
(* P.S. This is also kind of how Unicode works! *)
```

Optional Bonus (5 points). *Do everything else first!*

Many programming languages provide a function called `printf` that takes a string containing special "format specifiers" followed by some arguments to format and print. For example, in Java we can use printf like so (string with format specifiers in bold):

```
System.out.printf("%s!\n %d\n %f\n", "hello world", 1, 2.0);
```

Which produces output:

```
hello world!
 1
 2.0
```

Functions like printf are very interesting because the total number of arguments they take and the types of those arguments depend on the **value** of their first argument (a string containing format specifiers). In the example above, the format:

```
"%s!\n %d\n %f\n"
```

means that we also need to pass this call to `printf` a string (because of the specifier `"%s"`), an integer (the `"%d"`), and a floating point number (the `"%f"`). Notice how printf "puts its argument into the format string" before printing (e.g., it also prints the "!", spaces, and newlines from its first string argument in the example call above).

In SML, every function takes one argument, so we cannot directly implement `printf`. That is a bummer.

However, SML does provide some more flexible functions for printing beyond the `print` function we have seen in lecture. Recall the type of `print`:

```
print : string -> unit
```

`print` takes a string and, well, prints it. SML's `TextIO` module provides some other functions and values, including these two which we will use below:

```
TextIO.stdOut : outstream
TextIO.output : outstream * string -> unit
```

Calling `TextIO.output (TextIO.stdOut, s)` is the same as calling `print s`.

MLton is an awesome SML compiler made by an amazing team of hackers. Like many hackers, they like `printf` because it is very convenient. They are sad that SML does not directly support `printf`.

The MLton team also maintains a very interesting wiki on the web. Consider this module adapted from an example in the MLton wiki.

```
structure Printf = struct
  fun id x = x
  fun ignore x = ()
  fun on (_, f) = f (fn p => p ()) ignore
  fun fprintf out f = f (out, id)
  val printf = fn z => fprintf TextIO.stdOut z

  fun one ((out, f), make) g =
    g (out, fn r =>
      f (fn p =>
        make (fn s =>
          r (fn () => (p (); TextIO.output (out, s))))))

  fun % x s = one (x, fn f => f s)
  fun spec to x = one (x, fn f => fn x => f (to x))
  val S = fn z => spec id z
  val D = fn z => spec Int.toString z
  val F = fn z => spec Real.toString z
end
```

Using this module, we can write an SML version of the Java code above:

```
(* open so we can just write printf instead of Printf.printf *)
open Printf

val () =
  printf S % "!\n " D % "\n " F % "\n"
    on "hello world" 1 2.0
```

This produces the same output as our Java example:

```
  hello world!
   1
   2.0
```

Of course we can write many other examples as well:

```
val () =
  printf % "(" D % ", " D % ")\n"
    on 1 2

fun prompt (name, acct) =
  printf % "Hello " S % "! Is your account number: " D % "?"
    on name acct

fun display (a, b, c) =
  printf % "{ name = " S % "\n"
         % ", acct = " D % "\n"
         % ", time = " F % "}\n"
    on a b c
```

(A) SML functions only take one argument, only return one result, and only have one type. Yet, the calls to printf we showed in the earlier examples seem to take any number of arguments of all different kinds of types. How is this possible?

Higher-order function lets a function take a function as an argument, and currying lets a function take many arguments, by returning a function (that can be different depending on the first argument). This lets you write printf.

(B) What is the type of `Printf.on` ?

'a * (((unit -> 'b) -> 'b) -> ('c -> unit) -> 'd) -> 'd

(C) What is the type of `Printf.fprintf` ?

'a -> ('a * ('b -> 'b) -> 'c) -> 'c

(D) What is the type of `display` ?

TextIO.vector * int * real -> unit

(E) What do you like most about CSE 341 Winter 2020 so far?

The TAs!