Name:____

CSE341 Spring 2019, Midterm Examination May 3, 2019

Please do not turn the page until 12:30.

Rules:

- The exam is closed-book, closed-note, etc. except for *one* side of one 8.5x11in piece of paper.
- **Please stop promptly at 1:20.**
- There are **100 points**, distributed **unevenly** among **6** questions (all with multiple parts):
- **The exam is printed double-sided.**

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (**23** points) This problem uses this datatype binding for trees where all the data is at the leaves and each leaf can hold an int or a string.

```
datatype tree = I of int | S of string | N of tree * tree
```

   (a) Give an example of a value of type `tree` that contains exactly three ints and zero strings.

   (b) Define a function `sum` of type `tree -> int` that computes the sum of all the ints contained in the tree (ignoring any strings).

   (c) Define a function `stringify` of type `tree -> tree` that returns a result that is like its argument except everywhere the argument has some `I i`, the result has a `S s` where `s` is the string representation of `i`. For example, `stringify (N(I 3, S "hi")) = N(S "3", S "hi")`. Use library function `Int.toString` of type `int -> string`, which returns the string representation of its argument.

   (d) Define a function `intify` of type `tree -> tree` that returns a result that is like its argument except everywhere the argument has some `S s` *and* the `s` is the string representation of a number, the result has an `I i` where `s` is the string representation of `i`. For example, `intify (N(I 3, S "hi")) = N(I 3, S "hi")` and `intify (N(S "3", S "hi")) = N(I 3, S "hi")`. Use library function `Int.fromString` of type `string -> int option`, which returns an option containing the number represented by the string or `NONE` if the string does not represent a number.

Solution:

(a) `N(N(I 0,I 0), I 0)` or `N(I 0, N(I 0, I 0))` where each `0` can be any int.

(b)
```
fun sum t =
  case t of
      I i => i
    | S _  => 0
    | N(t1,t2) => sum t1 + sum t2
```

(c)
```
fun stringify t =
  case t of
      I i => S (Int.toString i)
    | S _ => t (* full credit for | S s => S s *)
    | N(t1,t2) => N(stringify t1, stringify t2)
```

(d)
```
fun intify t =
  case t of
      I _ => t (* full credit for | I i => I i *)
    | S s => (case Int.fromString s of
                  NONE => t (* or S s *)
                | SOME i => I i)
    | N(t1,t2) => N(intify t1, intify t2)
```

Name:____

More room if needed for Problem 1.

Name:____

2. (**17** points) This problem uses the same datatype binding as problem 1 and an *incorrect* function that is *supposed* to evaluate to true if and only if its argument contains exactly two ints.

```
datatype tree = I of int | S of string | N of tree * tree

fun has_exactly_two_ints t = (* this is buggy! *)
        case t of
(* 1 *)     N(I _, I _) => true
(* 2 *)    | N(t1, t2) => has_exactly_two_ints t1 orelse has_exactly_two_ints t2
(* 3 *)    | S _ => false
(* 4 *)    | I _ => false
```

(a) Give an example `tree` where `has_exactly_two_ints` evaluates to `true` correctly, i.e., the `tree` has exactly two ints.

(b) Give an example `tree` where `has_exactly_two_ints` evaluates to `true` incorrectly, i.e., the `tree` does not have exactly two ints.

(c) Give an example `tree` where `has_exactly_two_ints` evaluates to `false` correctly, i.e., the `tree` does not have exactly two ints.

(d) Give an example `tree` where `has_exactly_two_ints` evaluates to `false` incorrectly, i.e., the `tree` has exactly two ints.

(e) For each of i–iii below, choose from the options A–D:

A. The function is still buggy in exactly the same way.
B. The function is still buggy but is not equivalent to the version above.
C. The function is now correct.
D. The function no longer type-checks.

Do *not* consider that the first branch of a case expression has no `|` character and the others do. That is, assume that syntactic detail is fixed.

i. The branch labeled `(* 2 *)` is moved to before line `(* 1 *)` (so the order is 2, 1, 3, 4).

ii. The branch labeled `(* 3 *)` is moved to before line `(* 1 *)` (so the order is 3, 1, 2, 4).

iii. The branch labeled `(* 4 *)` is moved to before line `(* 1 *)` (so the order is 4, 1, 2, 3).

Solution:

(a) Must contain exactly one `N(I x, I y)` and no other `I z`.
(b) Must contain at least one `N(I x, I y)` and at least one other `I z`.
(c) Must contain no `N(I x, I y)` and have 0, 1, or strictly more than 2 `I x`. Examples: `I 0`, `S ""`, `N(I 0, N(I 0, N(I 0, S "")))`.
(d) Must contain no `N(I x, I y)` and have exactly 2 `I x`. Examples: `N(I 0, N(I 0, S ""))`, `N(N(I 0, S ""),N(I 0, S ""))`.
(e) i. D (the line labeled 1 is now an unreachable branch)
ii. A
iii. A

By the way, the question did not ask for a correct solution. Probably the most natural way is a helper function that counts how many ints there are:

```
fun has_exactly_two_ints t =
  let
      fun count_ints t =
        case t of
            I _ => 1
          | S _ => 0
          | N(t1,t2) => count_ints t1 + count_ints t2
  in
      count_ints t = 2
  end
```

This slightly different version "stops" examining more subtrees once it finds more than two ints:

```
fun has_exactly_two_ints t =
  let
      fun count_ints t =
        case t of
            I _ => 1
          | S _ => 0
          | N(t1,t2) =>
            let
                val x = count_ints t1
            in
                if x > 2 then x else x + count_ints t2
            end
  in
      count_ints t = 2
  end
```

Name:____

3. (**12** points) This problem asks you to give *good* error messages for why a short ML program does *not* type-check or has a syntax error. 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)
   ```
   fun f1 (x,y) =
       let
           val y = x
           val x = g 9
           fun g z = x * y
       in
           x + y + (g 7)
       end
   ```

   (b)
   ```
   fun f2 (x,y,z) =
       if x=y
       then 0
       else if y=z
       then 2
       else if y > 3
       then 3
   ```

   (c)
   ```
   fun f3 xs =
       case xs of
           [] => 0::[]
         | x::[] => x+1
         | x::xs => (x+1)::(f3 xs)
   ```

   (d)
   ```
   fun f4 (x,y) =
       if x > 0.0 orelse x = y
       then y
       else 3.0
   ```

   Solution:

   (a) The use of `g` in `g 9` is before the definition of `g` — there is no `g` in the environment here.

   (b) There is a syntax error because the last conditional expression has no else branch.

   (c) The first two branches of the case expression have incompatible types for the expressions: `int list` and `int`

   (d) `x` must have type `real` due to the comparison with `0.0`, but `real` is not an eqtype so `x = y` cannot type-check.

Name:____

4. (**18** points) Consider this ML function

```
fun foo f g xs =
    case xs of
        [] => []
      | x::xs' =>
        let
            val (y1,i1) = f x
            val (y2,i2) = g x
        in
            (if i1 > i2 then y1 else y2)::(foo f g xs')
        end
```

(a) What does `foo (fn x => (0,x*2)) (fn x => (x+1,x+1)) [0,2,1,3]` evaluate to?

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

(c) Complete this alternate definition of `foo` so that it is equivalent to the function above. (Hint: you need several short lines of code.)

```
fun foo f g = List.map (fn x =>





                       )
```

(d) What is the type of `foo (fn x => (x,x)) (fn x => (0-x,0-x))` ?

(e) In approximately one English sentence, what does the function produced by `foo (fn x => (x,x)) (fn x => (0-x,0-x))` compute?

Solution:

(a) `[1,0,2,0]`

(b) `('a -> 'b * int) -> ('a -> 'b * int) -> 'a list -> 'b list` (We announced during the exam to assume for this problem that `>` takes arguments of type `int`. Otherwise, the type could in principle also have `real` in both positions that `int` appears above, but ML would not actually make this choice without an explicit type annotation.)

(c)
```
fun foo  f g = List.map (fn x =>
                          let
                              val (y1,i1) = f x
                              val (y2,i2) = g x
                          in
                              if i1 > i2 then y1 else y2
                          end)
```

(d) `int list -> int list`

(e) It maps the absolute-value function across a list.

Name:____

5. (**9** points) Recall `List.filter` has type `('a -> bool) -> 'a list -> 'a list`. Consider this function of type `('a -> bool) -> ('a -> bool) -> 'a list -> 'a list`:

```
fun filter2 f g xs =
    case xs of
        [] => []
      | x::xs => if f x andalso g x
                 then x :: (filter2 f g xs)
                 else filter2 f g xs
```

(a) Is the recursive call in the then-branch above a tail call?

(b) Is the recursive call in the else-branch above a tail call?

(c) Is the call to `f` in the body of `filter2` a tail call?

(d) Is the call to `g` in the body of `filter2` a tail call?

(e) Reimplement `filter2` by filling in the blank below:

```
fun filter2 f g = List.filter ________________________________________
```

Solution:

(a) No
(b) Yes
(c) No
(d) No
(e) `fun filter2 f g = List.filter (fn x => f x andalso g x)`

Name:____

6. (**21** points) This problem considers two ML modules `Direction1` and `Direction2`, and a signature `DIRECTION`. They are on the next page. Separate that page from your exam and do *not* turn it in.

   Provided information: Under the signature `DIRECTION`, the two modules *are* equivalent to each other.

   (a) If we deleted the `turn` function in `Direction2` and replaced it with a copy of the `turn` function in `Direction1`, would the program (A) no longer type-check, (B) still type-check and have equivalent modules, or (C) type-check but with modules not equivalent?

   (b) If we deleted the `turn` function in `Direction1` and replaced it with a copy of the `turn` function in `Direction2`, would the program (A) no longer type-check, (B) still type-check and have equivalent modules, or (C) type-check but with modules not equivalent?

   (c) Given signature `DIRECTION`, consider client code outside of module `Direction2` (for the rest of Problem 6, `Direction1` is irrelevant).

      i. If possible, fill in the blank so that `nope` evaluates to `false`. If impossible, write "impossible."

```
val x = __________________
val nope = Direction2.isNS x orelse Direction2.isEW x
```

      ii. If possible, fill in the blank so `isWest` correctly implements a function that evaluates to `true` if and only if its argument is `Direction2`'s representation of west. If impossible, write "impossible."

```
fun isWest x = ________________
```

      iii. If possible, fill in the blank so `yep` evaluates to `true`. If impossible, write "impossible."

```
val west = _________________
val yep = Direction2.isEW west
```

   (d) Repeat part (c) but assuming the first line of `DIRECTION` is `type t = int`:

      i. If possible, fill in the blank so that `nope` evaluates to `false`. If impossible, write "impossible."

```
val x =  ___________________
val nope = Direction2.isNS x orelse Direction2.isEW x
```

      ii. If possible, fill in the blank so `isWest` correctly implements a function that evaluates to `true` if and only if its argument is `Direction2`'s representation of west. If impossible, write "impossible."

```
fun isWest x = ________________
```

      iii. If possible, fill in the blank so `yep` evaluates to `true`. If impossible, write "impossible."

```
val west = _________________
val yep = Direction2.isEW west
```

   (e) Repeat part (c) with `t` again abstract but assuming the line `val north : t` is removed.

      i. If possible, fill in the blank so that `nope` evaluates to `false`. If impossible, write "impossible."

```
val x = ___________________
val nope = Direction2.isNS x orelse Direction2.isEW x
```

      ii. If possible, fill in the blank so `isWest` correctly implements a function that evaluates to `true` if and only if its argument is `Direction2`'s representation of west. If impossible, write "impossible."

```
fun isWest x = ________________
```

iii. If possible, fill in the blank so `yep` evaluates to `true`. If impossible, write “impossible.”

```
val west = ________________
val yep = Direction2.isEW west
```

Solution:

The exam given in class had an incorrect defintion of `Direction2.turn`. As a result, we had to grade part (a) and (c,ii) and (d,ii) a bit more leniently than the sample solutions below. This posted version of the file fixed this problem.

(a) (B) type-checks and modules equivalent

(b) (A) no longer type-checks

(c)
   i. impossible
   ii. impossible
   iii. `Direction2.turn(Direction2.north,~1)` (other answers possible, including ones that do east instead of west)

(d)
   i. any `int` other than 0, 1, 2, 3 (e.g., 4 :-))
   ii. `x=3`
   iii. `3` (or technically `1`), (or `Direction2.turn(Direction2.north(~1))` or similar)

(e)
   i. impossible
   ii. impossible
   iii. impossible

These definitions are used in Problem 6. Rip this page out from the rest of the exam. Do not put answers on this page and do not turn it in.

```
signature DIRECTION =
sig
    type t
    val turn : t * int -> t
    val north : t
    val isNS : t -> bool
    val isEW : t -> bool
end

structure Direction1 :> DIRECTION =
struct

datatype t = North | East | South | West

fun turnClockwise x = case x of North => East | East => South | South => West | West => North

fun turnCounterClockwise x = case x of North => West | West => South | South => East | East => North

fun turn (x,n) =
    if n = 0
    then x
    else if n > 0
    then turn(turnClockwise x, n-1)
    else turn(turnCounterClockwise x, n+1)

val north = North

fun isNS x =
    case x of
        North => true
      | South => true
      | _ => false

fun isEW x =
    case x of
        East => true
      | West => true
      | _ => false
end

structure Direction2 :> DIRECTION =
struct

type t = int (* 0 = North, 1 = East, 2 = South, 3 = West *)

fun turnClockwise x = if x=3 then 0 else x+1
fun turnCounterClockwise x = if x=0 then 3 else x-1

fun turn (x,n) = (x+n) mod 4

val north = 0

fun isNS x = (x = 0) orelse (x=2)
fun isEW x = (x = 1) orelse (x=3)

end
```